Allinea DDT and MAP User Guide

Version v4.2.2-39977

Contents

Contents 1

1 Introduction 9
- 1.1 Allinea DDT . . . 9
- 1.2 Allinea MAP . . . 10
- 1.3 Purchasing . . . 10
- 1.4 Online Resources . . . 11
- 1.5 Obtaining Help . . . 11

2 Installation 12
- 2.1 Linux/Unix Installation . . . 12
  - 2.1.1 Graphical Install . . . 12
  - 2.1.2 Text-mode Install . . . 13
- 2.2 Mac Installation . . . 14
- 2.3 Windows Installation . . . 14
- 2.4 Licence Files . . . 15
- 2.5 Floating Licences . . . 15

3 Connecting to a Remote System 16
- 3.1 Remote Launch Settings . . . 16
- 3.2 Remote Script . . . 17
- 3.3 Using X Forwarding or VNC . . . 17

4 DDT: Starting 20
- 4.1 Running a Program . . . 22
  - 4.1.1 Application . . . 22
  - 4.1.2 MPI . . . 23
  - 4.1.3 OpenMP . . . 23
  - 4.1.4 CUDA . . . 23
  - 4.1.5 UPC . . . 24
    - 4.1.5.1 GCC UPC . . . 24
    - 4.1.5.2 Berkeley UPC . . . 24
  - 4.1.6 Memory Debugging . . . 24
  - 4.1.7 Environment Variables . . . 24
  - 4.1.8 Plugins . . . 24
- 4.2 `remote-exec` Required By Some MPIs . . . 25
- 4.3 Debugging Single-Process Programs . . . 26
- 4.4 Debugging OpenMP Programs . . . 26
- 4.5 Manual Launching of Multi-Process Non-MPI programs . . . 27
- 4.6 Debugging MPMD Programs . . . 28
  - 4.6.1 Debugging MPMD Programs in Compatibility mode . . . 29
- 4.7 Opening Core Files . . . 29
- 4.8 Attaching To Running Programs . . . 30
  - 4.8.1 Automatically Detected MPI Jobs . . . 30
  - 4.8.2 Attaching To A Subset Of An MPI Job . . . 31
  - 4.8.3 Manual Process Selection . . . 31
  - 4.8.4 Configuring Attaching to Remote Hosts . . . 32
  - 4.8.5 Using DDT Command-Line Arguments . . . 33
- 4.9 Starting A Job In A Queue . . . 33

© 2015 Allinea Software Ltd.

4.10 Using Custom MPI Scripts . . . 34
4.11 Starting DDT From A Job Script . . . 36

5 DDT: Overview 38
5.1 Saving And Loading Sessions . . . 39
5.2 Source Code . . . 39
5.3 Project Files . . . 39
5.3.1 Application / External Code . . . 39
5.4 Finding Lost Source Files . . . 40
5.5 Finding Code Or Variables . . . 40
5.5.1 Find File or Function . . . 40
5.5.2 Find . . . 40
5.5.3 Find in Files . . . 41
5.6 Jump To Line / Jump To Function . . . 41
5.7 Static Analysis . . . 42
5.8 Editing Source Code . . . 43
5.9 Version Control Information . . . 43

6 DDT: Controlling Program Execution 45
6.1 Process Control And Process Groups . . . 45
6.1.1 Detailed View . . . 45
6.1.2 Summary View . . . 46
6.2 Focus Control . . . 47
6.2.1 Overview of changing focus . . . 47
6.2.2 Process Group Viewer . . . 47
6.2.3 Breakpoints . . . 47
6.2.4 Code Viewer . . . 48
6.2.5 Parallel Stack View . . . 48
6.2.6 Playing and Stepping . . . 48
6.2.7 Step Threads Together . . . 48
6.2.8 Stepping Threads Window . . . 48
6.3 Hotkeys . . . 50
6.4 Starting, Stopping and Restarting a Program . . . 50
6.5 Stepping Through A Program . . . 50
6.6 Stop Messages . . . 51
6.7 Setting Breakpoints . . . 51
6.7.1 Using the Source Code Viewer . . . 51
6.7.2 Using the Add Breakpoint Window . . . 51
6.7.3 Pending Breakpoints . . . 52
6.7.4 Conditional Breakpoints . . . 53
6.8 Suspending Breakpoints . . . 53
6.9 Deleting A Breakpoint . . . 53
6.10 Loading And Saving Breakpoints . . . 54
6.11 Default Breakpoints . . . 54
6.12 Synchronizing Processes . . . 54
6.13 Setting A Watchpoint . . . 55
6.14 Tracepoints . . . 56
6.14.1 Setting a Tracepoint . . . 56
6.14.2 Tracepoint Output . . . 56
6.15 Version Control Breakpoints and Tracepoints . . . 57
6.16 Examining The Stack Frame . . . 59
6.17 Align Stacks . . . 59

© 2015 Allinea Software Ltd.

6.18 ``Where are my processes?" - Viewing Stacks in Parallel . . . 60
6.18.1 Overview . . . 60
6.18.2 The Parallel Stack View in Detail . . . 60
6.19 Browsing Source Code . . . 62
6.20 Simultaneously Viewing Multiple Files . . . 63
6.21 Signal Handling . . . 63
6.21.1 Custom Signal Handling (Signal Dispositions) . . . 64
6.21.2 Sending Signals . . . 64

7 DDT: Viewing Variables And Data 66
7.1 Sparklines . . . 66
7.2 Current Line . . . 66
7.3 Local Variables . . . 66
7.4 Arbitrary Expressions And Global Variables . . . 67
7.4.1 Fortran Intrinsics . . . 68
7.4.2 Changing the language of an Expression . . . 68
7.4.3 Macros and #defined Constants . . . 68
7.5 Help With Fortran Modules . . . 68
7.6 Viewing Complex Numbers in Fortran . . . 69
7.7 C++ STL Support . . . 70
7.8 Custom Pretty Printers . . . 70
7.8.1 Example . . . 70
7.9 Viewing Array Data . . . 71
7.10 UPC Support . . . 71
7.11 Changing Data Values . . . 72
7.12 Viewing Numbers In Different Bases . . . 72
7.13 Examining Pointers . . . 72
7.14 Multi-Dimensional Arrays in the Variable View . . . 72
7.15 Multi Dimensional Array Viewer (MDA) . . . 73
7.15.1 Array Expression . . . 74
7.15.2 Filtering by Value . . . 75
7.15.3 Distributed Arrays . . . 75
7.15.4 Advanced: How Arrays Are Laid Out in the Data Table . . . 75
7.15.5 Auto Update . . . 78
7.15.6 Statistics . . . 78
7.15.7 Export . . . 78
7.15.8 Visualization . . . 78
7.16 Cross-Process and Cross-Thread Comparison . . . 79
7.17 Assigning MPI Ranks . . . 81
7.18 Viewing Registers . . . 81
7.19 Interacting Directly With The Debugger . . . 82

8 DDT: Program Input And Output 83
8.1 Viewing Standard Output And Error . . . 83
8.2 Displaying Selected Processes . . . 83
8.3 Saving Output . . . 83
8.4 Sending Standard Input . . . 83

9 DDT: Logbook 85
9.1 Usage . . . 85
9.2 Comparison Window . . . 86

© 2015 Allinea Software Ltd.

10 DDT: Message Queues **87**
10.1 Viewing The Message Queues . . . 87
10.2 Interpreting the Message Queues . . . 88
10.3 Deadlock . . . 89

11 DDT: Memory Debugging **90**
11.1 Enabling Memory Debugging . . . 90
11.2 CUDA Memory Debugging . . . 90
11.3 Configuration . . . 91
11.3.1 Static Linking . . . 92
11.3.2 Available Checks . . . 93
11.3.3 Changing Settings at Run Time . . . 93
11.4 Pointer Error Detection and Validity Checking . . . 94
11.4.1 Library Usage Errors . . . 94
11.4.2 View Pointer Details . . . 94
11.4.3 Writing Beyond An Allocated Area . . . 96
11.4.4 Fencepost Checking . . . 96
11.4.5 Suppressing an Error . . . 96
11.5 Current Memory Usage . . . 97
11.5.1 Detecting Leaks when using Custom Allocators/Memory Wrappers . . . 98
11.6 Memory Statistics . . . 98

12 DDT: Checkpointing **100**
12.1 What Is Checkpointing? . . . 100
12.2 How To Checkpoint . . . 100
12.3 Restoring A Checkpoint . . . 100

13 DDT: Using and Writing Plugins **101**
13.1 Supported Plugins . . . 101
13.2 Installing a Plugin . . . 101
13.3 Using a Plugin . . . 101
13.4 Writing a Plugin . . . 102
13.5 Plugin Reference . . . 103

14 DDT: CUDA GPU Debugging **105**
14.1 Licensing . . . 105
14.2 Preparing to Debug GPU Code . . . 105
14.3 Launching the Application . . . 105
14.4 Controlling GPU threads . . . 106
14.4.1 Breakpoints . . . 106
14.4.2 Stepping . . . 106
14.4.3 Running and Pausing . . . 107
14.5 Examining GPU Threads and Data . . . 107
14.5.1 Selecting GPU Threads . . . 107
14.5.2 Viewing GPU Thread Locations . . . 108
14.5.3 Understanding Kernel Progress . . . 108
14.5.4 Source Code Viewer . . . 109
14.6 GPU Devices Information . . . 109
14.7 Attaching to running GPU applications . . . 109
14.8 Known Issues / Limitations . . . 110
14.8.1 Debugging Multiple GPU processes (CUDA 4.0 and below) . . . 110
14.8.2 Using Multiple GPU processes (CUDA 4.1 and above) . . . 110

© 2015 Allinea Software Ltd.

14.8.3 Thread control . . . 110
14.8.4 General . . . 110
14.8.5 Pre sm_20 GPUs . . . 111
14.8.6 Workaround for unsupported gcc versions with nvcc . . . 111
14.8.7 Debugging Multiple GPU processes on Cray limitations . . . 112
14.9 GPU Language Support . . . 112
14.9.1 CAPS HMPP . . . 112
14.9.2 Cray OpenACC . . . 112
14.9.3 PGI Accelerators and CUDA Fortran . . . 113

15 DDT: Offline Debugging 114
15.1 Using Offline Debugging . . . 114
15.2 Offline Report Output (HTML) . . . 115
15.3 Offline Report Output (Plain Text) . . . 116

16 DDT: Using DDT with the VisIt Visualization Tool 117
16.1 Support for VisIt . . . 117
16.2 Patching and Building VisIt . . . 117
16.3 Compatibility . . . 117
16.4 Enabling VisIt Support in DDT . . . 117
16.5 Setting Visualization Points (Vispoints) . . . 118
16.6 Using Vispoints in DDT . . . 120
16.7 Returning to DDT . . . 120
16.8 Focusing on a Domain & VisIt Picks . . . 121
16.9 Using DDT with a pre-instrumented program . . . 122

17 MAP: Starting 123
17.1 Preparing a Program for Profiling . . . 123
17.1.1 Debugging Symbols . . . 124
17.1.2 .eh-frame-hdr section . . . 124
17.1.3 Linking . . . 124
17.1.4 Static Linking . . . 124
17.1.5 Static Linking on Cray X-Series Systems . . . 126
17.1.6 Manual Dynamic Linking on Cray X-Series systems . . . 127
17.2 Profiling a Program . . . 128
17.2.1 Application . . . 128
17.2.2 MPI . . . 128
17.2.3 Environment Variables . . . 129
17.2.4 Profiling . . . 129
17.3 `remote-exec` Required By Some MPIs . . . 130
17.4 Profiling Single-Process Program . . . 131
17.5 Sending Standard Input . . . 131
17.6 Starting A Job In A Queue . . . 132
17.7 Using Custom MPI Scripts . . . 132
17.8 Starting MAP From A Job Script . . . 135
17.9 MAP Environment Variables . . . 135

18 MAP: Program Output 137
18.1 Viewing Standard Output And Error . . . 137
18.2 Displaying Selected Processes . . . 137
18.3 Saving Output . . . 137

© 2015 Allinea Software Ltd.

19 MAP: Source Code View **138**

20 MAP: Parallel Stack View **139**

21 MAP: Project Files View **140**

22 MAP: Metrics View **141**
22.1 Detecting MPI imbalance . . . 143

23 Running MAP from the Command Line **145**
23.1 Profiling MPMD Programs . . . 145

24 Configuration **146**
24.1 Configuration files . . . 146
24.1.1 Site Wide Configuration . . . 146
24.1.2 Converting Legacy Site-Wide Configuration Files . . . 146
24.1.3 Using Shared Home Directories on Multiple Systems . . . 147
24.1.4 Using a Shared Installation on Multiple Systems . . . 147
24.1.5 Importing Legacy Configuration . . . 147
24.2 Integration With Queuing Systems . . . 148
24.3 Template Tutorial . . . 149
24.3.1 The Template Script . . . 149
24.3.2 Configuring Queue Commands . . . 149
24.3.3 Configuring How Job Size is Chosen . . . 150
24.3.4 Quick Restart . . . 150
24.4 Connecting to remote programs (remote-exec) . . . 150
24.5 Optional Configuration . . . 151
24.5.1 System . . . 151
24.5.2 Job Submission . . . 152
24.5.3 Code Viewer Settings . . . 152
24.5.4 Appearance . . . 153
24.5.5 VisIt . . . 153

25 The Licence Server **154**
25.1 Running The Server . . . 154
25.2 Running DDT/MAP Clients . . . 154
25.3 Logging . . . 155
25.4 Troubleshooting . . . 155
25.5 Adding A New Licence . . . 155
25.6 Examples . . . 156
25.7 Example Of Access Via A Firewall . . . 156
25.8 Querying Current Licence Server Status . . . 157
25.9 Licence Server Handling Of Lost DDT/MAP Clients . . . 158

A Supported Platforms **159**
A.1 DDT . . . 159
A.2 MAP . . . 160

B MPI Distribution Notes and Known Issues **161**
B.1 Bull MPI . . . 161
B.2 HP MPI . . . 161
B.3 Intel MPI . . . 161
B.4 MPICH 2 . . . 162

© 2015 Allinea Software Ltd.

B.5 MPICH 3 . . . 162
B.6 IBM PE . . . 162
B.7 MVAPICH 1 . . . 162
B.8 MVAPICH 2 . . . 163
B.9 Open MPI . . . 163
B.10 SGI Altix / SGI MPT . . . 164
B.11 Cray MPT . . . 164
B.11.1 Using DDT with Cray ATP (the Abnormal Termination Process) . . . 164
B.12 Berkeley UPC . . . 165
B.13 SLURM . . . 165

C Compiler Notes and Known Issues 166
C.1 AMD OpenCL compiler . . . 166
C.2 Berkeley UPC Compiler . . . 166
C.3 Cray Compiler Environment . . . 166
C.4 GNU . . . 166
C.4.1 GNU UPC . . . 167
C.5 IBM XLC/XLF . . . 167
C.6 Intel Compilers . . . 167
C.7 Pathscale EKO compilers . . . 168
C.8 Portland Group Compilers . . . 169

D Platform Notes and Known Issues 171
D.1 CRAY . . . 171
D.2 GNU/Linux Systems . . . 172
D.2.1 General . . . 172
D.2.2 SUSE Linux . . . 172
D.3 IBM AIX Systems . . . 172
D.4 IBM Blue Gene/Q . . . 173
D.4.1 Attaching . . . 173
D.5 Intel Xeon Phi . . . 173
D.5.1 Requirements . . . 173
D.5.2 Installation . . . 174
D.5.3 Configuration . . . 174

E General Troubleshooting and Known Issues 179
E.1 General Troubleshooting . . . 179
E.1.1 Problems Starting the GUI . . . 179
E.1.2 Problems Reading this document . . . 179
E.2 Starting a Program . . . 179
E.2.1 Problems Starting Scalar Programs . . . 179
E.2.2 Problems Starting Multi-Process Programs . . . 180
E.2.3 No Shared Home Directory . . . 180
E.2.4 DDT/MAP says it can't find your hosts or the executable . . . 180
E.2.5 The progress bar doesn't move and DDT/MAP 'times out' . . . 181
E.2.6 The progress bar gets close to half the processes connecting and then stops and DDT/MAP 'times out' . . . 181
E.3 Attaching . . . 181
E.3.1 The system does not allow attaching to processes (Ubuntu) . . . 181
E.3.2 The system does not allow attaching to processes (Fedora, Red Hat) . . . 181
E.3.3 Running processes don't show up in the attach window . . . 182
E.4 Source Viewer . . . 182

© 2015 Allinea Software Ltd.

E.4.1 No variables or line number information . . . 182
E.4.2 Source code does not appear when you start DDT / MAP . . . 182
E.5 Input/Output . . . 182
E.5.1 Output to stderr is not displayed . . . 182
E.6 Controlling a Program . . . 183
E.6.1 Program jumps forwards and backwards when stepping through it . . . 183
E.6.2 DDT sometimes stop responding when using the Step Threads Together option . 183
E.7 Evaluating Variables . . . 183
E.7.1 Some variables cannot be viewed when the program is at the start of a function . 183
E.7.2 Incorrect values printed for Fortran array . . . 183
E.7.3 Evaluating an array of derived types, containing multiple-dimension arrays . . . 183
E.7.4 C++ STL types are not pretty printed . . . 184
E.7.5 The Fortran Module Browser is missing . . . 184
E.8 Memory Debugging . . . 184
E.8.1 The View Pointer Details window says a pointer is valid but doesn't show you which line of code it was allocated on . . . 184
E.8.2 `mprotect fails` error when using memory debugging with guard pages . . 184
E.8.3 Allocations made before or during `MPI_Init` show up in *Current Memory Usage* but have no associated stack back trace . . . 184
E.8.4 Deadlock when calling `printf` or `malloc` from a signal handler . . . 184
E.8.5 Program runs more slowly with Memory Debugging enabled . . . 185
E.9 MAP specific issues . . . 185
E.9.1 My compiler is inlining functions . . . 185
E.9.2 MPI Wrapper Libraries . . . 185
E.9.3 I'm not getting enough samples . . . 186
E.9.4 I just see main (external code) and nothing else . . . 186
E.9.5 MAP is reporting time spent in a function definition . . . 186
E.9.6 MAP is not correctly identifying vectorized instructions . . . 186
E.9.7 MAP harmless error messages in Xeon Phi . . . 186
E.9.8 MAP takes an extremely long time to gather and analyze my OpenBLAS-linked application . . . 187
E.10 Obtaining Support . . . 187

F Queue Template Script Syntax 188
F.1 Queue Template Tags . . . 188
F.2 Defining New Tags . . . 189
F.3 Specifying Default Options . . . 191
F.4 Launching . . . 191
F.4.1 Using AUTO_LAUNCH_TAG . . . 191
F.4.2 Using ddt-mpirun . . . 192
F.4.3 MPICH 1 based MPI . . . 192
F.4.4 Scalar Programs . . . 193
F.5 Using PROCS_PER_NODE_TAG . . . 193
F.6 Job ID Regular Expression . . . 193

1 Introduction

Welcome to the Allinea MAP and Allinea DDT user guide. Allinea MAP is our new, low-overhead profiler for both scalar and MPI programs. Allinea DDT is our industry-leading parallel debugger supporting a wide range of parallel architectures and models, including MPI, UPC, CUDA and OpenMP.

Both these tools share a common environment - one download provides you with everything you need to run both Allinea DDT and Allinea MAP, although your licence may restrict you to one or the other. This simplifies your installation and maintenance overheads and provides one common, familiar interface across all our tools, making it easy to move between them while working on a piece of code.

Note to cluster owners: you'll notice that some parts of this document are shared between both Allinea MAP and Allinea DDT, in particular the installation and configuration sections. Typically, both tools should be provided from one binary installation with one cluster-wide configuration shared between the two. This makes it as easy as possible for users of the cluster to switch between the tools without having to look up settings and reconfigure their queue submission scripts.

1.1 Allinea DDT

Allinea DDT is an intuitive, scalable, graphical debugger capable of debugging a wide variety of scenarios found in today's development environments. With Allinea DDT, it is possible to debug:

- Single process and multithreaded software
- OpenMP
- Parallel (MPI) software
- Heterogeneous software such as that written to use GPUs
- Hybrid codes mixing paradigms such as MPI + OpenMP, or MPI + CUDA
- Multi-process software of any form, including client-server applications

The tool can do many tasks beyond the normal capabilities of a debugger -- for example the memory debugging feature is able to detect some errors before they have caused a program crash by verifying usage of the system allocator functions, and the message queue integration with MPI can show the current state of communication between processes in the system.

Allinea DDT supports all of the compiled languages that are found in mainstream and high-performance computing including:

- C, C++, and all derivatives of Fortran, including Fortran 90.
- Parallel languages/models including MPI, UPC, and Fortran 2008 Co-arrays.
- GPU languages such as HMPP, OpenMP Accelerators, CUDA and CUDA Fortran.

Whilst many users choose Allinea DDT for desktop development or for debugging on small departmental parallel machines, it is also scalable and fast to beyond Petascale and is used to debug hundreds of thousands of processes simultaneously at some sites.

Chapters 4 to 16 of this manual describe DDT in more detail.

© 2015 Allinea Software Ltd.

1.2 Allinea MAP

Allinea MAP is a parallel profiler that aims to be powerful but easy to use, scaleable to hundreds of thousands of processes while maintaining a low overhead. Allinea MAP features:

- A sampling profiler with adaptive sampling rates to keep the data volumes collected under control. Samples are aggregated at all levels to preserve key features of a run without drowning in data
- A folding code and stack viewer allows you to drill down to time spent on individual lines and draw back to see the big picture across nests of routines.
- Just 5% application slowdown even with thousands of MPI proceesss.
- Both interactive and batch modes for gathering profile data.
- A unified job control interface with Allinea DDT -- configure it once and both tools just work, with a familiar, usable interface.
- Metrics that allow memory usage, floating-point calculations and MPI usage to be seen through a program run and across processes:
  - Flick to the CPU view to see the percentage of vectorized SIMD instructions, including AVX extensions used in each part of the code
  - See how the amount of time spent in memory operations varies over time and processes - are you making efficient use of the cache?
  - Zoom in to any part of the timeline, isolate a single iteration and explore its behaviour in detail
  - Everything shows aggregated data, preferring distributions with outlying ranks labelled to endless lists of processes and threads, ensuring the display is as visually scalable as our industry-leading backend.

Chapters 17 to 23 of this manual describe MAP in more detail.

1.3 Purchasing

To purchase a licence and support for either Allinea DDT or Allinea MAP, contact sales@allinea.com or visit http://www.allinea.com/products/allinea-ddt/purchase or http://www.allinea.com/products/allinea-map/purchase to purchase online.

There are a number of different licence types, which determine the possible usage scenarios.

- Workstation Scalar -- for single process or multi-threaded code, including unlimited thread counts. Locked to a single workstation.
- Workstation Parallel -- for single process, multi-threaded, multi-process or parallel code up to 8 distinct processes and unlimited thread counts. Locked to a workstation.
- Cluster -- for all types of software, up to a defined process count and maximum number of concurrent users. The user interface is locked to one machine (but may still be X-forwarded) but the parallel processes may run on other machines.
- Supercomputing -- a more flexible licence for all types of software, floating up to a defined total number of concurrent processes in use by multiple users concurrently.
- Extreme -- our most flexible licence, able to support multiple architectures and floating similar to the Supercomputing licence.

© 2015 Allinea Software Ltd.

Additionally, CUDA support is an option which can be added to any DDT licence to allow debugging of GPU software for NVIDIA CUDA devices. CUDA kernels cannot be profiled in MAP at this time

Evaluation licences contain support for all the features of Allinea DDT or MAP but are limited to 16 processes. Allinea DDT licences include a permanent MAP trial licence that allows profiling data to be collected for 30 seconds at the full scale of the DDT licence.

1.4 Online Resources

You can find links to tutorials, training material, webinars and white papers in our online knowledge center:

Knowledge Center http://www.allinea.com/help-and-resources/training

Known issues and the latest version of this user guide may be found on the support web pages:

Support http://www.allinea.com/knowledge-center/get-support

1.5 Obtaining Help

Whilst this document attempts to cover as many parts of the installation, features and use of our tool as possible, there will be scenarios or configurations that are not covered, or are only briefly mentioned, or you may on occasion experience a problem using the product. In any event, the support team at Allinea will be able to help and will look forward to assist in ensuring that you can get the most out of Allinea DDT and MAP.

You can contact the team by sending an email directly to support@allinea.com .

Please provide as much detail as you can about the scenario in hand, such as:

- Version number of Allinea DDT or MAP and your operating system and the distribution (example: Red Hat Enterprise Linux 5.8). This information is all available by using the -v option to DDT or MAP on the command line:

```
bash$ ddt -v

Allinea DDT
Part of the Allinea environment.
(c) Allinea Software 2002-2014

Version: 4.2.1
Build: Ubuntu 12.04 x86_64
Build Date: Feb 29 2014

Licence Serial Number: see About window

Frontend OS: Ubuntu 12.04 x86_64
Nodes' OS: unknown
Last connected ddt-debugger: unknown
```

- The compiler used and version number
- The MPI library and CUDA toolkit version as appropriate.

© 2015 Allinea Software Ltd.

2 Installation

A combined release of Allinea DDT and MAP (the Allinea Tools) may be downloaded from the Allinea website http://www.allinea.com. Follow the instructions below to install it.

2.1 Linux/Unix Installation

2.1.1 Graphical Install

Untar the package and run the `installer` executable using the commands below.

```
tar xf allinea-tools-<unknown>-ARCH.tar
cd allinea-tools-<unknown>-ARCH
./installer
```

The installer consists of a number of pages where you can choose install options. Use the *Next* and *Back* buttons to move between pages or *Cancel* to cancel the installation.

The *Install Type* page lets you choose who you want to install DDT and MAP for. If you are an administrator (`root`) you may install the tools for *All Users* in a common directory such as `/opt` or `/usr/local`, otherwise only the *Just For Me* option is enabled.

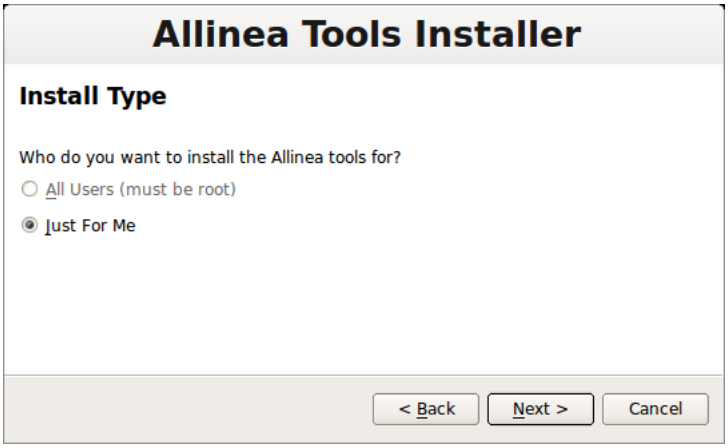

Allinea Tools Installer
Install Type
Who do you want to install the Allinea tools for?
All Users (must be root)
Just For Me
< Back
Next >
Cancel

Figure 1: *Allinea Tools Installer - Installation type*

Once you have selected the installation type, you will be asked what directory you would like to install the tools in. If you are installing on a cluster, make sure you choose a directory that is shared between the cluster login node / frontend and the cluster nodes. Otherwise you must install or copy it to the same location on each node.

© 2015 Allinea Software Ltd.

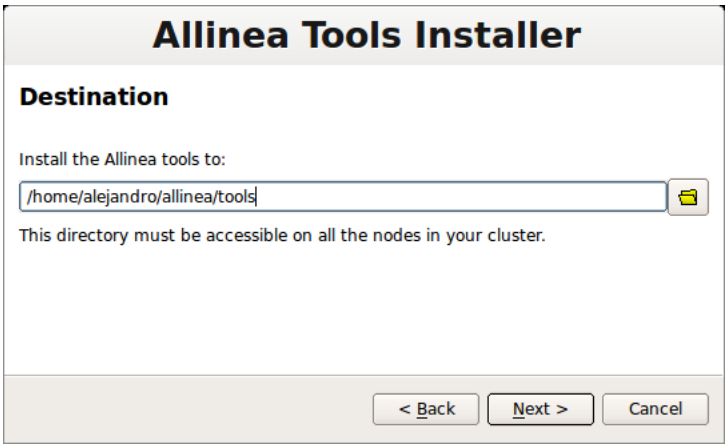


Figure 2: *Installer - Installation directory*

You will be shown the progress of the installation on the *Install* page.

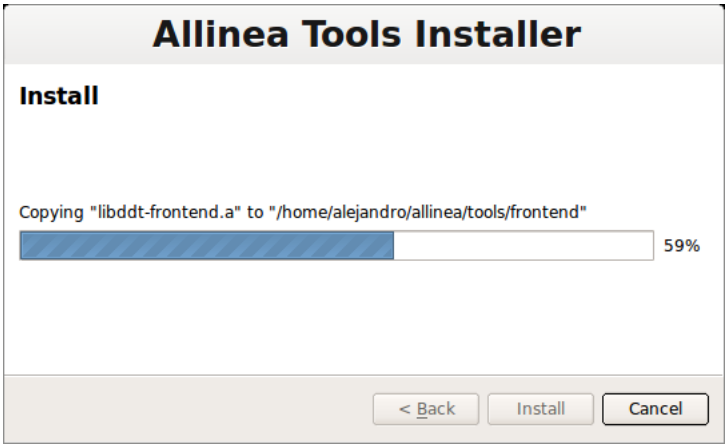


Figure 3: *Install in progress*

Icons for DDT and MAP will be added to your desktop environment's *Development* menu.

It is important to follow the instructions in the README file that is contained in the tar file. In particular, you will need a valid licence file. You can obtain an evaluation licence by completing the form at http://www.allinea.com/products/allinea-ddt/free-trial.

Due to the vast number of different site configurations and MPI distributions that are supported by Allinea DDT and MAP, it is inevitable that sometimes you may need to take further steps to get the tools fully integrated into your environment. For example, it may be necessary to ensure that environment variables are propagated to remote nodes, and ensure that the tool libraries and executables are available on the remote nodes.

2.1.2 Text-mode Install

The text-mode install script `textinstall.sh` is useful if you are installing remotely.

```
tar xf allinea-tools-<unknown>-ARCH.tar
cd allinea-tools-<unknown>-ARCH
./text-install.sh
```

© 2015 Allinea Software Ltd.

Press *Return* to read the licence when prompted and then enter the directory where you would like to install the tools. The directory must be accessible on all the nodes in your cluster.

2.2 Mac Installation

The Allinea Tools client for Mac is supplied as an Apple Disk Image (*.dmg) file. This contains a copy of this user guide, the release notes, and the Allinea Tools Client app bundle (i.e. Allinea Tools Client 4.2.app). This bundle must be drag and dropped into the chosen installation directory.

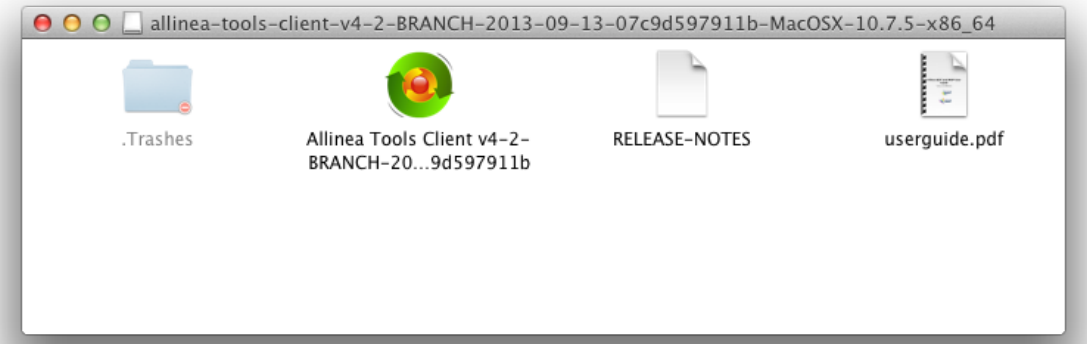


Figure 4: *Mac Allinea Tools Installer -- Installation Folder*

2.3 Windows Installation

The Allinea Tools client for Windows is installed using a graphical installer. This is a familiar Windows set-up executable, although care needs to be taken with the choice of a destination folder for the installation.

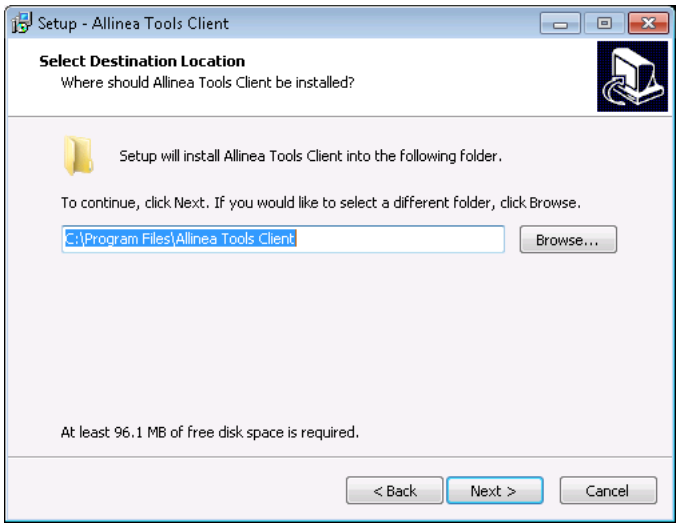

Setup - Allinea Tools Client
Select Destination Location
Where should Allinea Tools Client be installed?
Setup will install Allinea Tools Client into the following folder.
To continue, click Next. If you would like to select a different folder, click Browse.
C:\Program Files\Allinea Tools Client
Browse...
At least 96.1 MB of free disk space is required.
< Back
Next >
Cancel

Figure 5: *Windows Allinea Tools Installer -- Installation Folder*

If the user performing the installation has administrative rights, then the default installation folder is

`C:\Program Files\Allinea Tools`. If administrative rights have not been granted, then the default will be `C:\Users\<user>\AppData\Local`.

2.4 Licence Files

You can have combined or individual licence files for DDT and MAP, stored in {*installation-directory*}`licences` (e.g. `/home/user/allinea/tools/licenses/Licence.ddt,Licence.map`).

If this is inconvenient, the user can specify the location of a licence file using an environment variable, ALLINEA_LICENCE_DIR. For example:

```
export ALLINEA_LICENCE_DIR=${HOME}/SomeOtherLicenceDir
```

The user also has the choice of using ALLINEA_LICENSE_`DIR` as the environment variable (American spelling). The older DDT_LICENCE_FILE / DDT_LICENSE_FILE name for a single DDT licence still works, but we suggest you change to the new system

The order of precedence when searching for licence files is:

- `${ALLINEA_LICENCE_DIR}/*`
- `${ALLINEA_LICENSE_DIR}/*`
- `${DDT_LICENCE_FILE}`
- `${DDT_LICENSE_FILE}`
- {*installation-directory*}`/licences`

If you do not have a licence file, the DDT GUI will not start and a warning message will be presented. For MAP the GUI will still allow you to view old profiles, but you will not be able to collect new ones.

Time-limited evaluation licences are available from the Allinea website: http://www.allinea.com.

2.5 Floating Licences

For users with floating licences, the licensing daemon must be started prior to running DDT or MAP. It is recommended that this is done as a non-root user -- such as `nobody` or a special unprivileged user created for this purpose.

{**installation-directory**}**/bin/licenceserver [path-to-licences-dir]&**

This will start the daemon, it will serve all floating licences in path-to-licences-dir. If no path is specified, the default {`installation-directory`}`/licences` is used.

The host name, port and MAC (network) address of the licence server will be agreed with you before issuing the licence, and you should ensure that the agreed host and port will be accessible by users.

DDT / MAP clients will use a separate client licence file which identifies the host, port, and licence number.

Log files can be generated for accounting purposes.

For more information on the Licence Server please see section 25 The Licence Server.

© 2015 Allinea Software Ltd.

3 Connecting to a Remote System

Often you will need to login to a remote system in order to run a job. For example you may use SSH to login from your desktop machine *mydesktop* to the login node *mycluster-login* and then start a job using the queue submission command *qsub*.

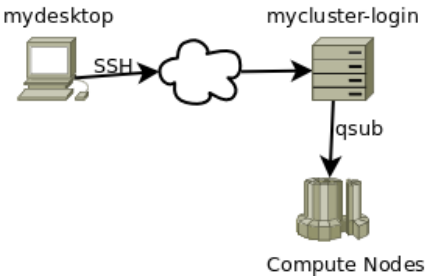


Figure 6: *Connecting to a Remote System*

The Allinea Tools can connect to remote systems using SSH for you so you can run the user interface on your desktop or laptop machine without the need for X forwarding. Native remote clients are available for Windows, Mac OS X and Linux.

No licence file is required by the remote client.

Note: The Allinea Tools must be installed on the remote system to use DDT or MAP remotely.

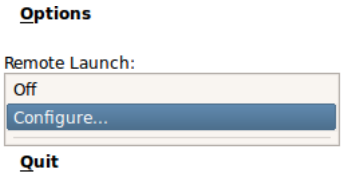


Figure 7: *Remote Launch -- Configure*

To connect to a remote system click on the *Remote Launch* drop down list and select *Configure...*. The *Remote Launch Settings* window will open where you can enter the necessary settings.

3.1 Remote Launch Settings

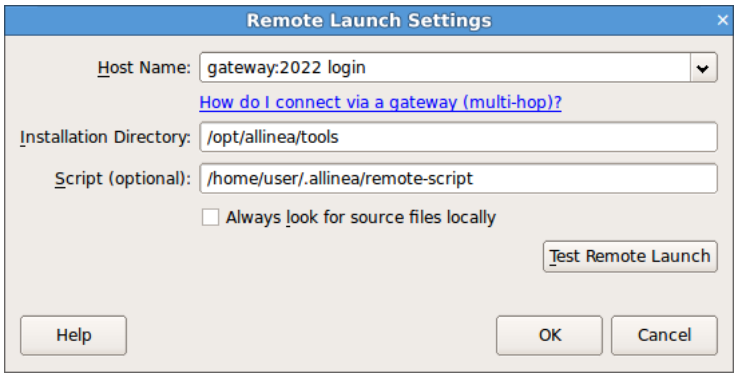


Figure 8: *Remote Launch Options*

Host Name: The host name of the remote system you wish to connect to.

The syntax of the host name field is:

```
[username]@hostname[:port]...
```

`username` is an optional user name to use on the remote system. If not specified your local user name will be used instead.

`hostname` is the host name of the remote system.

`port` is the optional port number that the remote host's SSH daemon is listening on. If not specified the default of 22 is used.

To login via one or more intermediate hosts (e.g. a gateway) enter the host names in order, separated by spaces, e.g. `gateway login`

Note: You must be able to login to the third and subsequent hosts without a password.

Additional SSH options may be specified in the `remote-exec` script covered in section ˜24.4 Connecting to remote programs (remote-exec).

Installation Directory: The full path to the Allinea Tools installation on the remote system.

Script (optional): This script will be run before starting the remote daemon on the remote system. You may use this script to load the required modules for DDT and MAP, your MPI and compiler. See below for more details.

Always look for source files locally: Check this box to use the source files on the local system instead of the remote system.

3.2 Remote Script

The script may load modules using the `module` command or otherwise set environment variables. The Allinea Tools will source this script before running its remote daemon (your script does not need to start the remote daemon itself).

The script will be run using `/bin/sh` (usually a Bourne-compatible shell). If this is not your usual login shell, make allowances for the different syntax it might require.

You may install a site-wide script that will be sourced for all users at `/path/to/allinea-tools/remote-init`.

Example Script

Note: this script file should be created on the remote system and the full path to the file entered in the Script (optional) *box.*

```
module load allinea-tools\\
module load mympi\\
module load mycompiler
```

3.3 Using X Forwarding or VNC

In the event you do not want to use the *Remote Launch* feature here are two other methods for running Allinea DDT or Allinea MAP on a remote system: X forwarding and VNC (or similar Unix-supporting remote desktop software).

© 2015 Allinea Software Ltd.

X forwarding is effective when the network connection is low latency (e.g. same physical site). VNC is *strongly* recommended when the network connection is moderate or slow.

- Apple users accessing a Linux or other Unix machine whilst using a single-button mouse should be advised that pressing the *Command* key and the single mouse button will have the same effect as right clicking on a two button mouse. Right clicking allows access to some important features in DDT and MAP.

  You can use X forwarding to access the Allinea Tools running on a remote Linux/Unix system from an Apple:

  – Start the X11 server (available in the `X11User.pkg`).

  – Set the display variable correctly to allow X applications to display by opening a terminal in OS/X and typing:

  ```
  export DISPLAY=:0
  ```

  – Then ssh to the remote system from that terminal, with ssh options -X and -C (X forwarding and compression). For example:

  ```
  ssh -CX username@login.mybigcluster.com
  ```

  – Now start DDT or MAP on the remote system and the window will appear on your Mac.

- Windows users can use any one of a number of commercial and open source X servers, but may find VNC a viable alternative (http://www.realvnc.com/) which is available under free and commercial licensing options.

- VNC allows users to access a desktop running on a remote server (e.g. a cluster login node or front end) and is more suitable than X forwarding for medium to high latency links. By setting up an SSH 'tunnel' users are usually able to securely access this remote desktop from anywhere. To use VNC and the Allinea Tools:

  – Log in to the remote system and set up a tunnel for port 5901 and 5801. On Apple or any Linux/Unix systems use the ssh command. If you are using Putty on Windows use the GUI to setup the tunnel.

  ```
  ssh -L 5901:localhost:5901 -L 5801:localhost:5801 \
  username@login.mybigcluster.com
  ```

  – At the remote prompt, start vncserver. If this is the first time you have used VNC it will ask you to set an access password.

  ```
  vncserver
  ```

  The output from vncserver will tell you which ports VNC has started on -- 5800+*n* and 5900+*n*, where *n* is the number given as hostname:n in the output. If this number, *n*, is not 1, then another user is already using VNC on that system, and you should set a new tunnel to these ports by logging in to the same host again and changing the settings to the new ports (or use SSH escape codes to add a tunnel, see the SSH manual pages for details).

  – Now, on the local desktop/laptop, either use a browser and access the desktop within the browser by entering the URL `http://localhost:5801/`, or (better) you may use a separate VNC client such as `krdc` or `vncviewer`.

  ```
  krdc localhost:1
  ```

  or

```
vncviewer localhost:1
```

If *n* is not 1, as described above, use :2, :3 etc. as appropriate instead.

- Note that a bug in the browser based access method means the *Tab* key does not work correctly in VNC. but `krdc` or `vncviewer` users are not affected by this problem.
- VNC frequently defaults to an old X window manager (twm) which requires you to manually place windows; this can be changed by editing the `~/.vnc/xstartup` file to use KDE or GNOME and restarting the VNC server.

© 2015 Allinea Software Ltd.

4 DDT: Starting

As always, when compiling the program that you wish to debug, you must add the debug flag to your compile command. For most compilers this is `-g`. It is also advisable to turn off compiler optimisations as these can make debugging appear strange and unpredictable. If your program is already compiled without debug information you will need to make the files that you are interested in again.

To start DDT simply type one of the following into a shell window:

```
ddt
ddt program_name
ddt program_name arguments
```

To start DDT in the Mac OS X type:

```
open '/Applications/Allinea Tools/Allinea DDT.app'
'/Applications/Allinea \
 Tools/Allinea DDT.app/Contents/MacOS/Allinea DDT'
[program_name [arguments]]
```

The quotes are used to cope with spaces in the path naming. The root `/Applications` directory is the default place for all applications in OS X. If your installation has been done in a user's home `~/Applications` directory then use that one.

Note: You should not attempt to pipe input directly to DDT – for information about how to achieve the effect of sending input to your program, please read section 8 DDT: Program Input And Output.

Once DDT has started it will display the Welcome Page.

© 2015 Allinea Software Ltd.

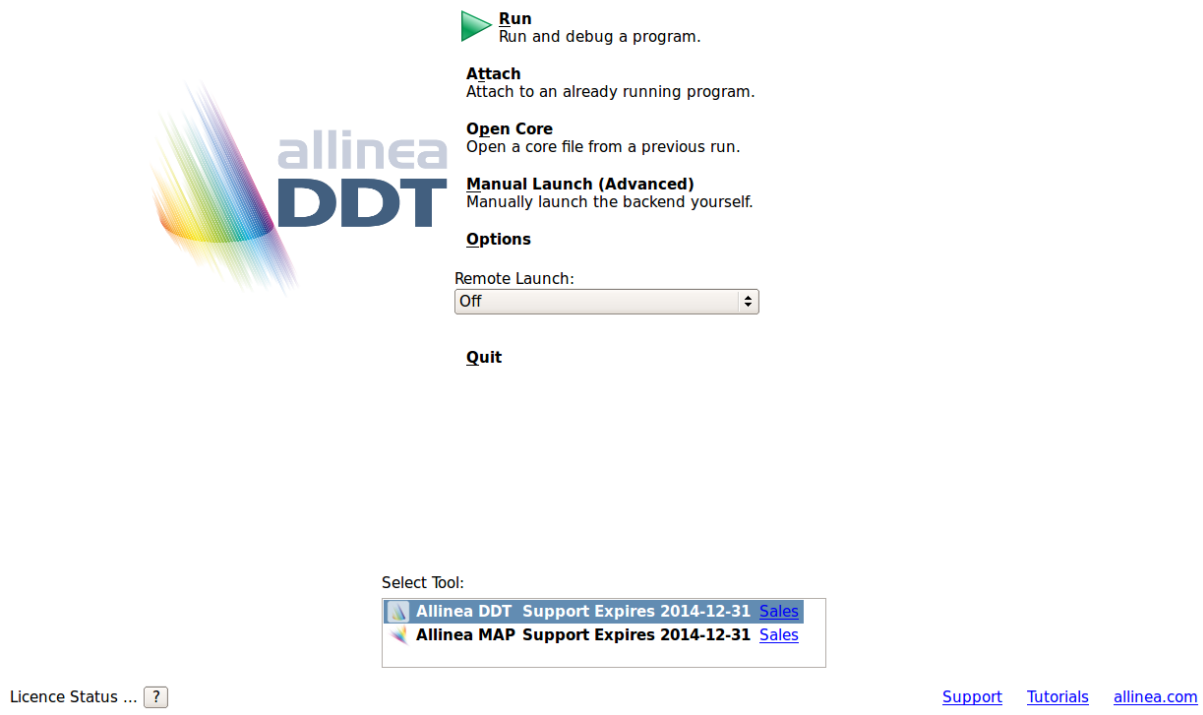


Figure 9: *DDT Welcome Page*

The *Welcome Page* allows you to choose what kind of debugging you want to do. You can:

- run a program from DDT and debug it
- debug a program you launch manually (e.g. on the command line)
- attach to an already running program
- open core files generated by a program that crashed
- connect to a remote system

© 2015 Allinea Software Ltd.

4.1 Running a Program

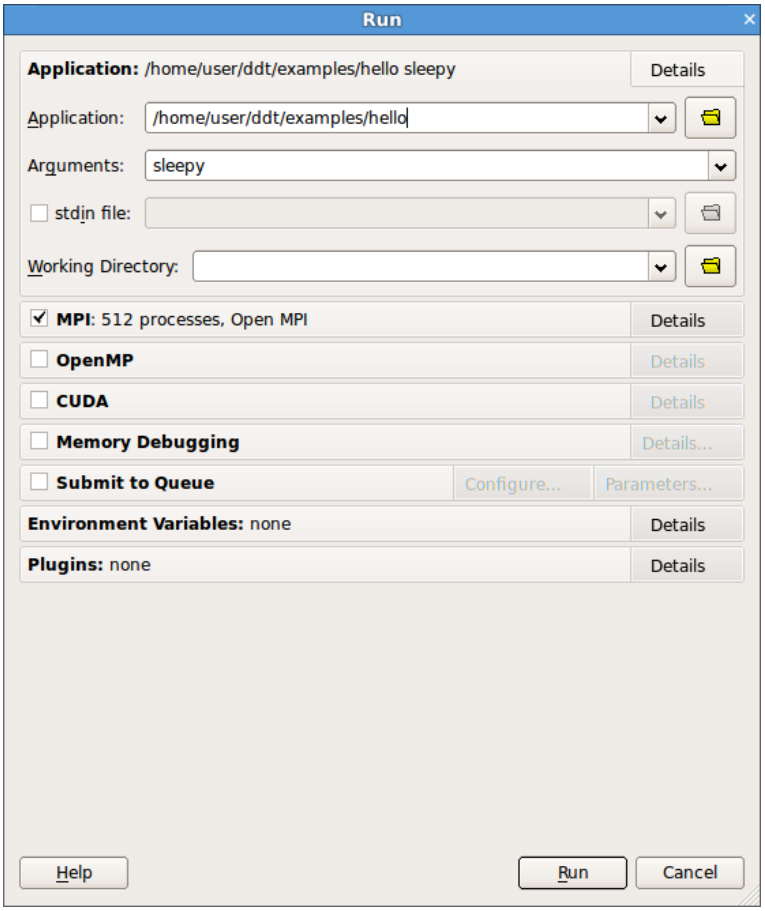


Figure 10: *Run Window*

If you click the *Run* button on the *Welcome Page* you will see the window above. The settings are grouped into sections. Click the *Details*... button to expand a section. The settings in each section are described below.

4.1.1 Application

Application: The full path name to your application. If you specified one on the command line, this will already be filled in. You may browse for an application by clicking on the *Browse* button.

Note: Many MPIs have problems working with directory and program names containing spaces. We recommend avoiding the use of spaces in directory and file names.

Arguments: (optional) The arguments passed to your application. These will be automatically filled if you entered some on the command line.

Note: Avoid using quote characters such as ' and “, as these may be interpreted differently by DDT and your command shell. If you must use these and cannot get them to work as expected, please contact support@allinea.com .

stdin file: (optional) This allows you to choose a file to be used as the standard input (`stdin`) for your program. DDT will automatically add arguments to `mpirun` to ensure your input file is used.

© 2015 Allinea Software Ltd.

Working Directory: (optional) The working (i.e. current directory) to use when debugging your application. If this is blank then DDT's working directory will be used instead.

4.1.2 MPI

Note: If you only have a single process licence or have selected none *as your* MPI Implementation *the* MPI *options will be missing. The* MPI *options are not available when DDT is in single process mode. See section 4.3 Debugging Single-Process Programs for more details about using DDT with a single process.*

Number of processes: The number of processes that you wish to debug. DDT supports hundreds of thousands of processes but this is limited by your licence. This option may not be displayed if disabled on the *Job Submission* options page.

Number of nodes: This is the number of compute nodes that you wish to use to run your program. This option is only displayed for certain MPI implementations or if it is enabled on the *Job Submission* options page.

Processes per node: This is the number of MPI processes to run on each compute node. This option is only displayed for certain MPI implementations or if it is enabled on the *Job Submission* options page.

Implementation: The MPI implementation to use. If you are submitting a job to a queue the queue settings will also be summarised here. You may change the MPI implementation by clicking on the *Change...* button.

Note: The choice of MPI implementation is critical to correctly starting DDT. Your system will normally use one particular MPI implementation. If you are unsure as to which to pick, try generic, consult your system administrator or Allinea. A list of settings for common implementations is provided in Appendix B MPI Distribution Notes and Known Issues .

Note: If your desired MPI command is not in your `PATH`*, or you wish to use an MPI run command that is not your default one, you can configure this using the* Options *window (See section 24.5.1 System).*

mpirun arguments: (optional): The arguments that are passed to `mpirun` or your equivalent – usually prior to your executable name in normal `mpirun`usage. You can place machine file arguments – if necessary – here. For most users this box can be left empty. You can also specify `mpirun`arguments on the command line (using the `-mpiargs` command line argument) or using an environment variable (using the `DDT_MPIRUN_ARGUMENTS` environment variable) if this is more convenient.

Note: You should ***not*** *enter the* `-np` *argument as DDT will do this for you.*

4.1.3 OpenMP

Number of OpenMP threads: The number of OpenMP threads to run your application with. The `OMP_NUM_THREADS` environment variable is set to this value.

4.1.4 CUDA

If your licence supports it, you may also debug GPU programs by enabling CUDA support. For more information on debugging CUDA programs, please see section 14 DDT: CUDA GPU Debugging.

Track GPU Allocations: Tracks CUDA memory allocations made using `cudaMalloc`, etc. See 11.2 CUDA Memory Debugging for more information.

Detect invalid read/writes: Turns on the CUDA-MEMCHECK error detection tool. See 11.2 CUDA Memory Debugging for more information.

4.1.5 UPC

The DDT configuration depends on the UPC compiler used.

4.1.5.1 GCC UPC

DDT can debug applications compiled with GCC UPC 4.8 with TLS disabled. See section C.4 GNU.

To run a UPC program in DDT you have to select the MPI implementation ``*GCC libupc SMP (no TLS)*''

4.1.5.2 Berkeley UPC

To run a Berkeley UPC program in DDT you have to compile the program using `-tv` flag and then select the same MPI implementation used in the Berkeley compiler build configuration.

The Berkeley compiler must be build using the MPI transport.

See section C.2 Berkeley UPC Compiler.

4.1.6 Memory Debugging

Clicking the *Details...* button will open the *Memory Debugging Settings* window. See section 11.3 Configuration for full details of the available Memory Debugging settings.

4.1.7 Environment Variables

The optional *Environment Variables* section should contain additional environment variables that should be passed to `mpirun`or its equivalent. These environment variables may also be passed to your program, depending on which MPI implementation your system uses. Most users will not need to use this box.

Note: on some systems it may be necessary to set environment variables for the DDT backend itself. For example: if `/tmp` is unusable on the compute nodes you may wish to set `TMPDIR` to a different directory. You can specify such environment variables in `/path/to/ddt/lib/environment`. Enter one variable per line and separate the variable name and value with `=`, e.g. `TMPDIR=/work/user`.

4.1.8 Plugins

The optional *Plugins* section allows you to enable plugins for various third-party libraries, such as the Intel Message Checker or Marmot. See section 13 DDT: Using and Writing Plugins for more information.

Click Run to start your program – or Submit if working through a queue (See section 24.2 Integration With Queuing Systems). This will run your program through the debug interface you selected and will allow your MPI implementation to determine which nodes to start which processes on.

Note: If you have a program compiled with Intel `ifort` or GNU `g77` you may not see your code and highlight line when DDT starts. This is because those compilers create a pseudo `MAIN` function,

above the top level of your code. To fix this you can either open your Source Code window and add a breakpoint in your code – then run to that breakpoint, or you can use the Step into function to step into your code.

When your program starts, DDT will attempt to determine the MPI world rank of each process. If this fails, you will see the following error message:

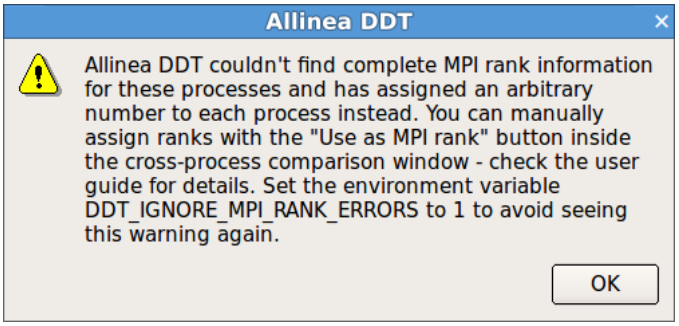

Allinea DDT
Allinea DDT couldn't find complete MPI rank information for these processes and has assigned an arbitrary number to each process instead. You can manually assign ranks with the "Use as MPI rank" button inside the cross-process comparison window - check the user guide for details. Set the environment variable DDT_IGNORE_MPI_RANK_ERRORS to 1 to avoid seeing this warning again.
OK

Figure 11: *MPI rank error*

This means that the number DDT shows for each process may not be the MPI rank of the process. To correct this you can tell DDT to use a variable from your program as the rank for each process – see section 7.17 Assigning MPI Ranks for details.

To end your current debugging session select the *End Session* menu option from the *File* menu. This will close all processes and stop any running code. If any processes remain you may have to clean them up manually using the `kill` command (or a command provided with your MPI implementation).

4.2 `remote-exec` Required By Some MPIs

When using *Open MPI*, *SGI MPT*, *MPICH 1 Standard* or the MPMD variants of *MPICH 2*, *MPICH 3* or *Intel MPI*, DDT will allow `mpirun`to start all the processes, then attach to them while they're inside `MPI_Init`.

This method is often faster than the generic method, but requires the `remote-exec` facility in DDT to be correctly configured if processes are being launched on a remote machine. For more information on `remote-exec`, please see section 24.4 Connecting to remote programs (remote-exec).

Important: If DDT is running in the background (e.g. `ddt &`) then this process may get stuck (some SSH versions cause this behaviour when asking for a password). If this happens to you, go to the terminal and use the `fg` or similar command to make DDT a foreground process, or run DDT again, without using "`&`".

If DDT can't find a password-free way to access the cluster nodes then you will not be able to use the specialised startup options. Instead, You can use *generic*, although startup may be slower for large numbers of processes.

© 2015 Allinea Software Ltd.

4.3 Debugging Single-Process Programs

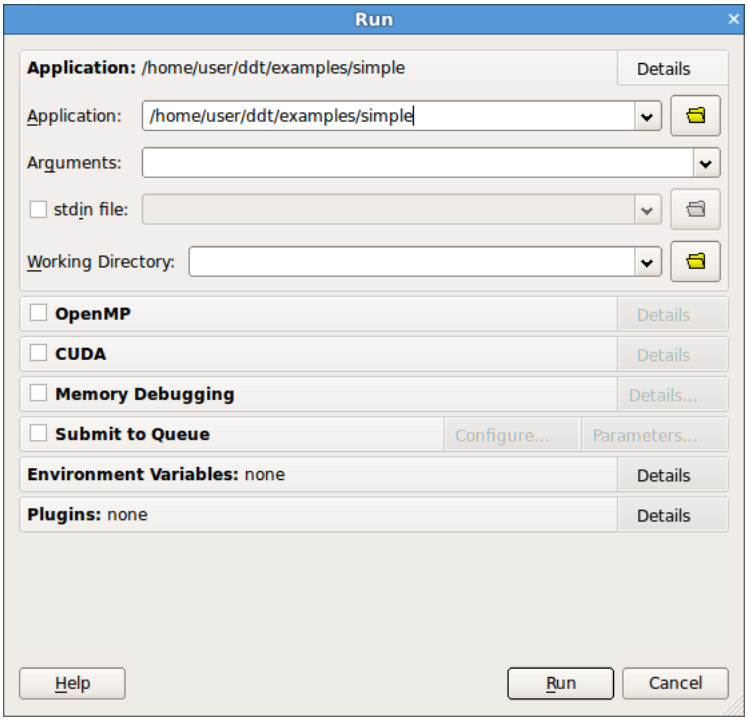


Figure 12: *Single-Process Run Window*

Users with single-process licences will immediately see the *Run Window* that is appropriate for single-process applications.

Users with multi-process licences can uncheck the *MPI* check box to run a single process program.

Select the application – either by typing the file name in, or selecting using the browser by clicking the browse button. Arguments can be typed into the supplied box.

Finally click *Run* to start your program.

Note: If you have a program compiled with Intel `ifort` *or GNU* `g77` *you may not see your code and highlight line when DDT starts. This is because those compilers create a pseudo* `MAIN` *function, above the top level of your code. To fix this you can either open your Source Code window and add a breakpoint in your code – then play to that breakpoint, or you can use the Step Into function to step into your code.*

To end your current debugging session select the End Session menu option from the File menu. This will close all processes and stop any running code.

4.4 Debugging OpenMP Programs

When running an OpenMP program, set the *Number of OpenMP threads* value to the number of threads you require. DDT will run your program with the `OMP_NUM_THREADS` environment variable set to the appropriate value.

There are several important points to keep in mind while debugging OpenMP programs:

1. Parallel regions created with `#pragma omp parallel` (C) or `!$OMP PARALLEL` (Fortran) will usually not be nested in the *Parallel Stack View* under the function that contained the

`#pragma`. Instead they will appear under a different top-level item. The top-level item is often in the OpenMP runtime code, and the parallel region appears several levels down in the tree.

2. Some OpenMP libraries only create the threads when the first parallel region is reached. Don't worry if you can only see one thread at the start of the program.
3. You cannot step into a parallel region. Instead, tick the *Step threads together* box and use the *Run to here* command to synchronise the threads at a point inside the region – these controls are discussed in more detail in their own sections of this document.
4. You cannot step out of a parallel region. Instead, use *Run to here* to leave it. Most OpenMP libraries work best if you keep the *Step threads together* box ticked until you have left the parallel region. With the Intel OpenMP library, this means you will see the *Stepping Threads* window and will have to click *Skip All* once.
5. Leave *Step threads together* off when you are outside a parallel region (as OpenMP worker threads usually do not follow the same program flow as the main thread).
6. To control threads individually, use the *Focus on Thread* control. This allows you to step and play one thread without affecting the rest. This is helpful when you want to work through a locking situation or to bring a stray thread back to a common point. The Focus controls are discussed in more detail in their own section of this document.
7. Shared OpenMP variables may appear twice in the Locals window. This is one of the many unfortunate side-effects of the complex way OpenMP libraries interfere with your code to produce parallelism. One copy of the variable may have a nonsense value – this is usually easy to recognise. The correct values are shown in the Evaluate and Current Line windows.
8. Parallel regions may be displayed as a new function in the stack views. Many OpenMP libraries implement parallel regions as automatically-generated “outline” functions, and DDT shows you this. To view the value of variables that are not used in the parallel region, you may need to switch to thread 0 and change the stack frame to the function you wrote, rather than the outline function.
9. Stepping often behaves unexpectedly inside parallel regions. Reduction variables usually require some sort of locking between threads, and may even appear to make the current line jump back to the start of the parallel region! Don't worry about this – step over another couple of times and you'll see it comes back to where it belongs.
10. Some compilers optimise parallel loops regardless of the options you specified on the command line. This has many strange effects, including code that appears to move backwards as well as forwards, and variables that have nonsense values because they have been optimised out by the compiler.

If you are using DDT with OpenMP and would like to tell us about experiences, we would appreciate your feedback. Please email support@allinea.com with the subject title *OpenMP feedback*.

4.5 Manual Launching of Multi-Process Non-MPI programs

DDT can only launch MPI programs and scalar (single process) programs itself. The *Manual Launch (Advanced)* button on the *Welcome Page* allows you to debug multi-process and multi-executable programs. These programs don't necessarily need to be MPI programs. You can debug programs that use other parallel frameworks, or both the client and the server from a client/server application in the same DDT session, for example.

You must run each program you want to debug manually using the `ddt-client` command, similar to debugging with a scalar debugger like the GNU debugger (gdb). However, unlike a scalar debugger, you

© 2015 Allinea Software Ltd.

can debug more than one process at the same time in the same DDT session (licence permitting). Each program you run will show up as a new process in the DDT window.

For example to debug both client and server in the same DDT session:

1. Click on the *Manual Launch (Advanced)* button.
2. Select 2 processes

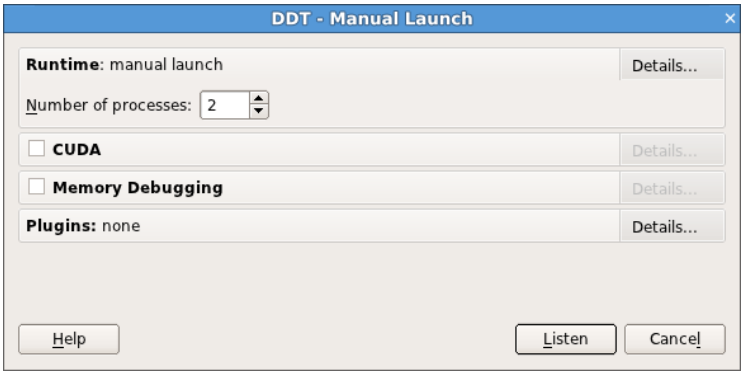


Figure 13: *Manual Launch Window*

3. Click the Listen button.
4. At the command line run:

```
ddt-client server &
ddt-client client &
```

The server process will appear as process 0 and the client as process 1 in the DDT window.

Figure 14: *Manual Launch Process Groups*

After you have run the initial programs you may add extra processes to the DDT session (for example extra clients) using ddt-client in the same way.

```
ddt-client client2 &
```

If you check *Start debugging after the first process connects* you do not need to specify how many processes you want to launch in advance. You can start debugging after the first process connects and add extra processes later as above.

4.6 Debugging MPMD Programs

If you are using Open MPI, MPICH 2, MPICH 3 or Intel MPI, DDT can be used to debug multiple program, multiple data (MPMD) programs. To start an MPMD program in DDT:

1. MPICH 2 and Intel MPI only: Select the MPMD variant of the MPI Implementation on the *System* page of the *Options* window, e.g. for MPICH 2 select *MPICH 2 (MPMD)*.

2. Click the *Run* button on the *Welcome Page*.
3. Select one of the MPMD programs in the *Application* box, it doesn't matter what executable you choose.
4. Enter the total amount of processes for the MPMD job in the *Number of processes* box.
5. Enter an MPMD style command line in the *mpirun Arguments* box in the MPI section of the *Run* window, e.g.:

   ```
   -np 4 hello : -np 4 program2
   ```

   or:

   ```
   --app /path/to/my_app_file
   ```

6. Click the Run button.

Note: be sure that the sum of processes in step 5 is equal to the number of processes set in step 4.

4.6.1 Debugging MPMD Programs in Compatibility mode

If you are using Open MPI in *Compatibility* mode (e.g. because you don't have SSH access to the compute nodes) then replace:

```
-np 2 ./progc.exe : -np 4 ./progf90.exe
```

in the *mpirun Arguments* / appfile with this:

```
-np 2 /path/to/ddt/bin/ddt-client ./progc.exe : -np 4
/path/to/ddt/bin/ddt-client ./progf90.exe
```

4.7 Opening Core Files

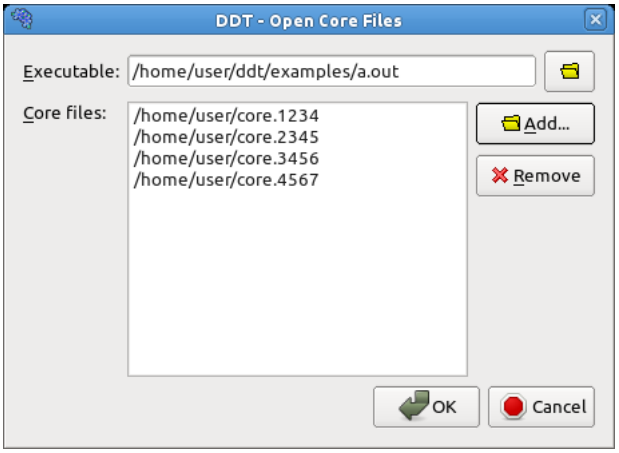


Figure 15: *The Open Core Files Window*

DDT allows you to open one or more core files generated by your application.

To debug using core files, click the *Open Core Files* button on the *Welcome Page*. This opens the *Open Core Files* window, which allows you to select an executable and a set of core files. Click *OK* to open the core files and start debugging them.

© 2015 Allinea Software Ltd.

While DDT is in this mode, you cannot play, pause or step (because there is no process active). You are, however, able to evaluate expressions and browse the variables and stack frames saved in the core files.

The *End Session* menu option will return DDT to its normal mode of operation.

4.8 Attaching To Running Programs

DDT can attach to running processes on any machine you have access to, whether they are from MPI or scalar jobs, even if they have different executables and source pathnames. Clicking the *Attach to a Running Program* button on the *Welcome Page* will show DDT's *Attach Window*:

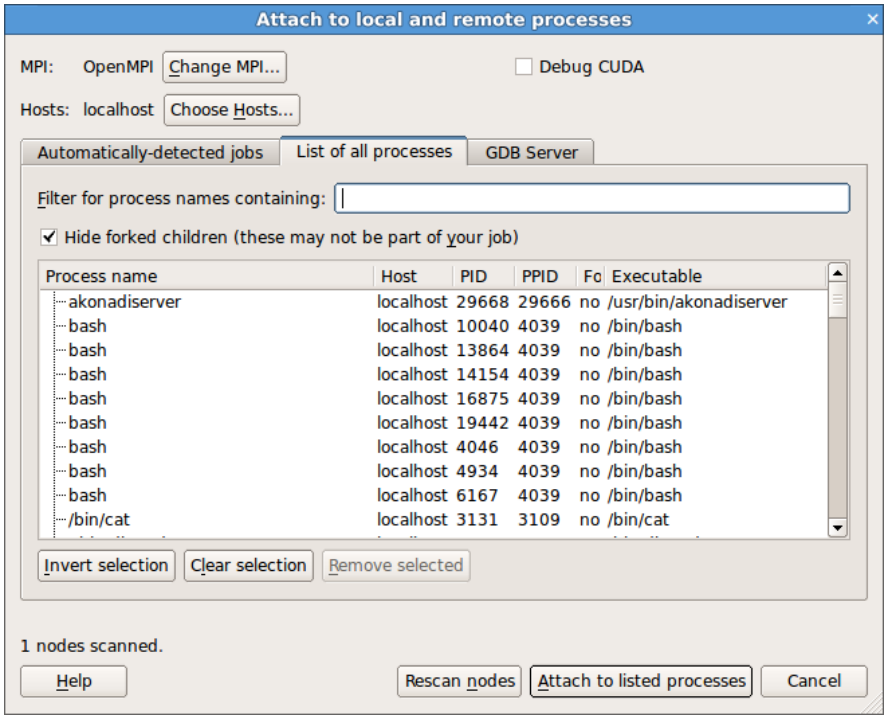

Attach to local and remote processes
MPI: OpenMPI Change MPI...
Debug CUDA
Hosts: localhost Choose Hosts...
Automatically-detected jobs
List of all processes
GDB Server
Filter for process names containing:
Hide forked children (these may not be part of your job)
Process name Host PID PPID Fo Executable
akonadiserver localhost 29668 29666 no /usr/bin/akonadiserver
bash localhost 10040 4039 no /bin/bash
bash localhost 13864 4039 no /bin/bash
bash localhost 14154 4039 no /bin/bash
bash localhost 16875 4039 no /bin/bash
bash localhost 19442 4039 no /bin/bash
bash localhost 4046 4039 no /bin/bash
bash localhost 4934 4039 no /bin/bash
bash localhost 6167 4039 no /bin/bash
/bin/cat localhost 3131 3109 no /bin/cat
Invert selection
Clear selection
Remove selected
1 nodes scanned.
Help
Rescan nodes
Attach to listed processes
Cancel

Figure 16: *Attach Window*

There are two ways to select the processes you want to attach to: you can either choose from a list of automatically detected MPI jobs (for supported MPI implementations) or manually select from a list of processes.

4.8.1 Automatically Detected MPI Jobs

DDT can automatically detect MPI jobs started on the local host for selected MPI implementations (and other hosts you have access to if an *Attach Hosts File* is configured – see section 24.5.1 System for more details).

The list of detected MPI jobs is shown on the *Automatically-detected MPI jobs* tab of the *Attach Window*. Click the header for a particular job to see more information about that job. Once you have found the job you want to attach to simply click the *Attach* button to attach to it.

© 2015 Allinea Software Ltd.

4.8.2 Attaching To A Subset Of An MPI Job

You may want to attach only to a subset of ranks from your MPI job. You can choose this subset using the *Attach* to ranks box on the *Automatically-detected MPI jobs* tab of the Attach Window. You may change the subset later by selecting the *File* → *Change Attached Processes...* menu item.

4.8.3 Manual Process Selection

You can manually select which processes to attach to from a list of processes using the *List of all processes* tab of the *Attach Window*. If you want to attach to a process on a remote host see section 24.4 Connecting to remote programs (remote-exec) first.

Initially the list of processes will be blank while DDT scans the nodes, provided in your node list file, for running processes. When all the nodes have been scanned (or have timed out) the window will appear as shown above. Use the Filter box to find the processes you want to attach to. On non-Linux platforms you will also need to select the application executable you want to attach to. Ensure that the list shows all the processes you wish to debug in your job, and no extra/unnecessary processes. You may modify the list by selecting and removing unwanted processes, or alternatively selecting the processes you wish to attach to and clicking on *Attach to Selected Processes*. If no processes are selected, DDT uses the whole visible list.

On Linux you may use DDT to attach to multiple processes running different executables. When you select processes with different executables the application box will change to read *Multiple applications selected*. DDT will create a process group for each distinct executable

With some supported MPI implementations (e.g. Open MPI) DDT will show MPI processes as children of the `mpirun`(or equivalent) command (see figure below). Clicking the `mpirun`command will automatically select all the MPI child processes.

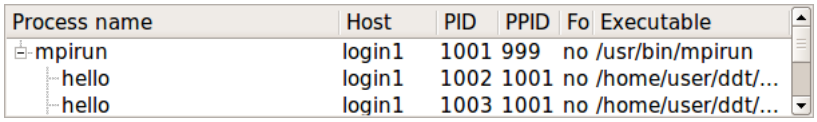


Figure 17: *Attaching with Open MPI*

Some MPI implementations (such as MPICH 1) create forked (child) processes that are used for communication, but are not part of your job. To avoid displaying and attaching to these, make sure the *Hide Forked Children* box is ticked. DDT's definition of a forked child is a child process that shares the parent's name. Some MPI implementations create your processes as children of each other. If you cannot see all the processes in your job, try clearing this checkbox and selecting specific processes from the list.

Once you click on the *Attach to Selected/Listed Processes* button, DDT will use `remote-exec` to attach a debugger to each process you selected and will proceed to debug your application as if you had started it with DDT. When you end the debug session, DDT will detach from the processes rather than terminating them – this will allow you to attach again later if you wish.

DDT will examine the processes it attaches to and will try to discover the `MPI_COMM_WORLD` rank of each process. If you have attached to two MPI programs, or a non-MPI program, then you may see the following message:

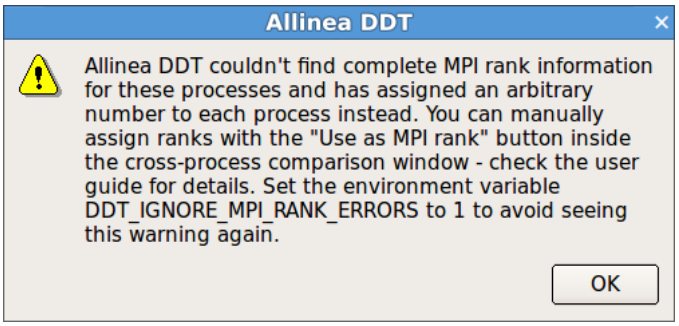

Allinea DDT
Allinea DDT couldn't find complete MPI rank information for these processes and has assigned an arbitrary number to each process instead. You can manually assign ranks with the "Use as MPI rank" button inside the cross-process comparison window - check the user guide for details. Set the environment variable DDT_IGNORE_MPI_RANK_ERRORS to 1 to avoid seeing this warning again.
OK

Figure 18: *MPI rank error*

If there is no rank (for example, if you've attached to a non-MPI program) then you can ignore this message and use DDT as normal. If there is, then you can easily tell DDT what the correct rank for each process via the *Use as MPI Rank* button in the *Cross-Process Comparison Window* – see section 7.17 Assigning MPI Ranks for details.

Note that the `stdin`, `stderr` and `stdout` (standard input, error and output) are not captured by DDT if used in attaching mode. Any input/output will continue to work as it did before DDT attached to the program (e.g. from the terminal or perhaps from a file).

4.8.4 Configuring Attaching to Remote Hosts

To attach to remote hosts in DDT, click the Choose Hosts button in the attach dialog. This will display the list of hosts to be used for attaching.

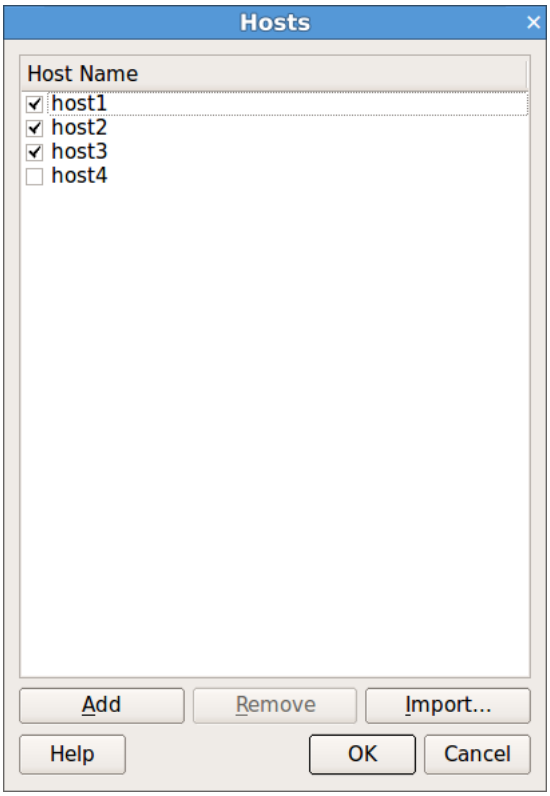


Figure 19: *Choose Hosts Window*

From here you can add and remove hosts, as well as unchecking hosts that you wish to temporarily

© 2015 Allinea Software Ltd.

exclude.

You can also import a list of hosts from a file by clicking the *Import* button.

The hosts list is initially populated from the *attach Hosts File*, which can be configured from the *Options window*: *File* → *Options* (*DDT* → *Preferences* on Mac OS X) .

Each remote host is then scanned for processes, and the result displayed in the attach window. If you have trouble connected to remote hosts, please see section 24.4 Connecting to remote programs (remote-exec).

4.8.5 Using DDT Command-Line Arguments

As an alternative to starting DDT and using the *Welcome Page*, DDT can instead be instructed to attach to running processes from the command-line.

To do so, you will need to specify the pathname to the application executable as well as a list of hostnames and process identifiers (PIDs).

The list of hostnames and PIDs can be given on the command-line using the `-attach` option:

```
mark@holly:~$ ddt -attach /home/mark/ddt/examples/hello \
   localhost:11057 \
   localhost:11094 \
   localhost:11352 \
   localhost:11362 \
   localhost:12357
```

Another command-line possibility is to specify the list of hostnames and PIDs in a file and use the `-attach-file` option:

```
mark@holly:~$ cat /home/mark/ddt/examples/hello.list

localhost:11057
localhost:11094
localhost:11352
localhost:11362
localhost:12357

mark@holly:~$ ddt -attach-file \
   /home/mark/ddt/examples/hello.list \
   /home/mark/ddt/examples/hello
```

In both cases, if just a number is specified for a `hostname`:`PID` pair, then `localhost`: is assumed.

These command-line options work for both single- and multi-process attaching.

4.9 Starting A Job In A Queue

If DDT has been configured to be integrated with a queue/batch environment, as described in section 24.2 Integration With Queuing Systems then you may use DDT to launch your job. In this case, a Submit button is presented on the *Run Window*, instead of the ordinary *Run* button. Clicking *Submit* from the *Run Window* will display the queue status until your job starts. DDT will execute the display command every second and show you the standard output. If your queue display is graphical or interactive then you cannot use it here.

© 2015 Allinea Software Ltd.

If your job does not start or you decide not to run it, click on *Cancel Job*. If the regular expression you entered for getting the job id is invalid or if an error is reported then DDT will not be able to remove your job from the queue – it is strongly recommend you check the job has been removed before submitting another as it is possible for a forgotten job to execute on the cluster and either waste resources or interfere with other debug sessions.

Once your job is running, it will connect to DDT and you will be able to debug it.

4.10 Using Custom MPI Scripts

On some systems a custom '`mpirun`' replacement is used to start jobs, such as `mpiexec`. DDT will normally use whatever the default for your MPI implementation is, so for MPICH 1 it would look for `mpirun`and not `mpiexec`. This section explains how to configure DDT to use a custom `mpirun`command for job start up.

There are typically two ways you might want to start jobs using a custom script, and DDT supports them both. Firstly, you might pass all the arguments on the command-line, like this:

```
mpiexec -n 4 /home/mark/program/chains.exe /tmp/mydata
```

There are several key variables in this line that DDT can fill in for you:

1. The number of processes (4 in the above example)
2. The name of your program (`/home/mark/program/chains.exe`)
3. One or more arguments passed to your program (`/tmp/mydata`)

Everything else, like the name of the command and the format of it's own arguments remains constant. To use a command like this in DDT, we adapt the queue submission system described in the previous section. For this `mpiexec`example, the settings would be as shown here:

© 2015 Allinea Software Ltd.

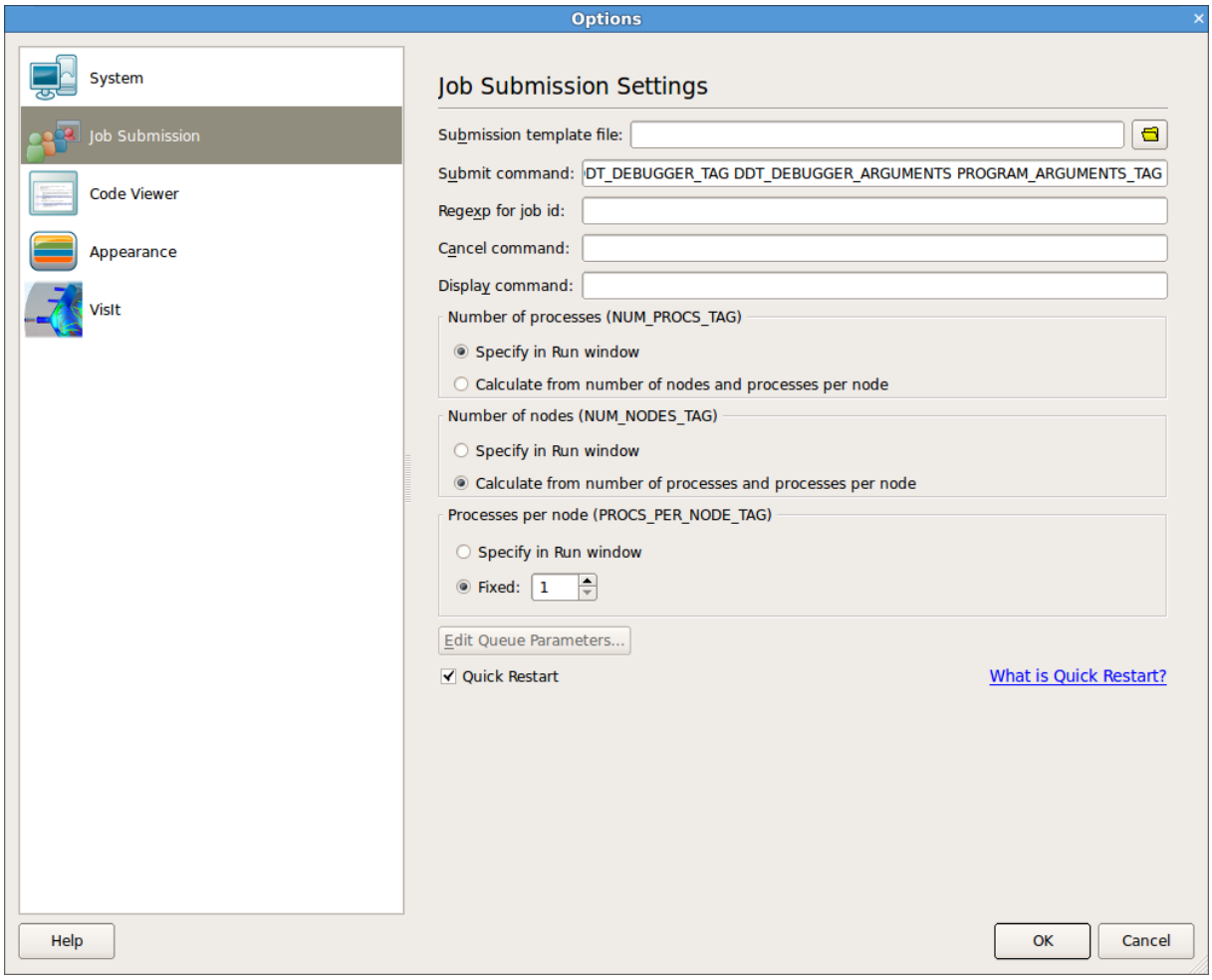


Figure 20: *DDT Using Custom MPI Scripts*

As you can see, most of the settings are left blank. Let's look at the differences between the *Submit Command* in DDT and what you would type at the command-line:

1. The number of processes is replaced with `NUM_PROCS_TAG`
2. The name of the program is replaced by the full path to `ddt-debugger`
3. The program arguments are replaced by `PROGRAM_ARGUMENTS_TAG`

Note, it is NOT necessary to specify the program name here. DDT takes care of that during its own startup process. The important thing is to make sure your MPI implementation starts `ddt-debugger` instead of your program, but with the same options.

The second way you might start a job using a custom `mpirun`replacement is with a settings file:

```
mpiexec -config /home/mark/myapp.nodespec
```

where myfile.nodespec might contains something like this:

```
comp00 comp01 comp02 comp03 : /home/mark/program/chains.exe /tmp/
   mydata
```

DDT can automatically generate simple configuration files like this every time you run your program – you just need to specify a template file. For the above example, the template file `myfile.ddt` would contain the following:

```
comp00 comp01 comp02 comp03 : DDTPATH_TAG/bin/ddt-debugger
   DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_ARGUMENTS_TAG
```

This follows the same replacement rules described above and in detail in section 24.2 Integration With Queuing Systems. The options settings for this example might be:

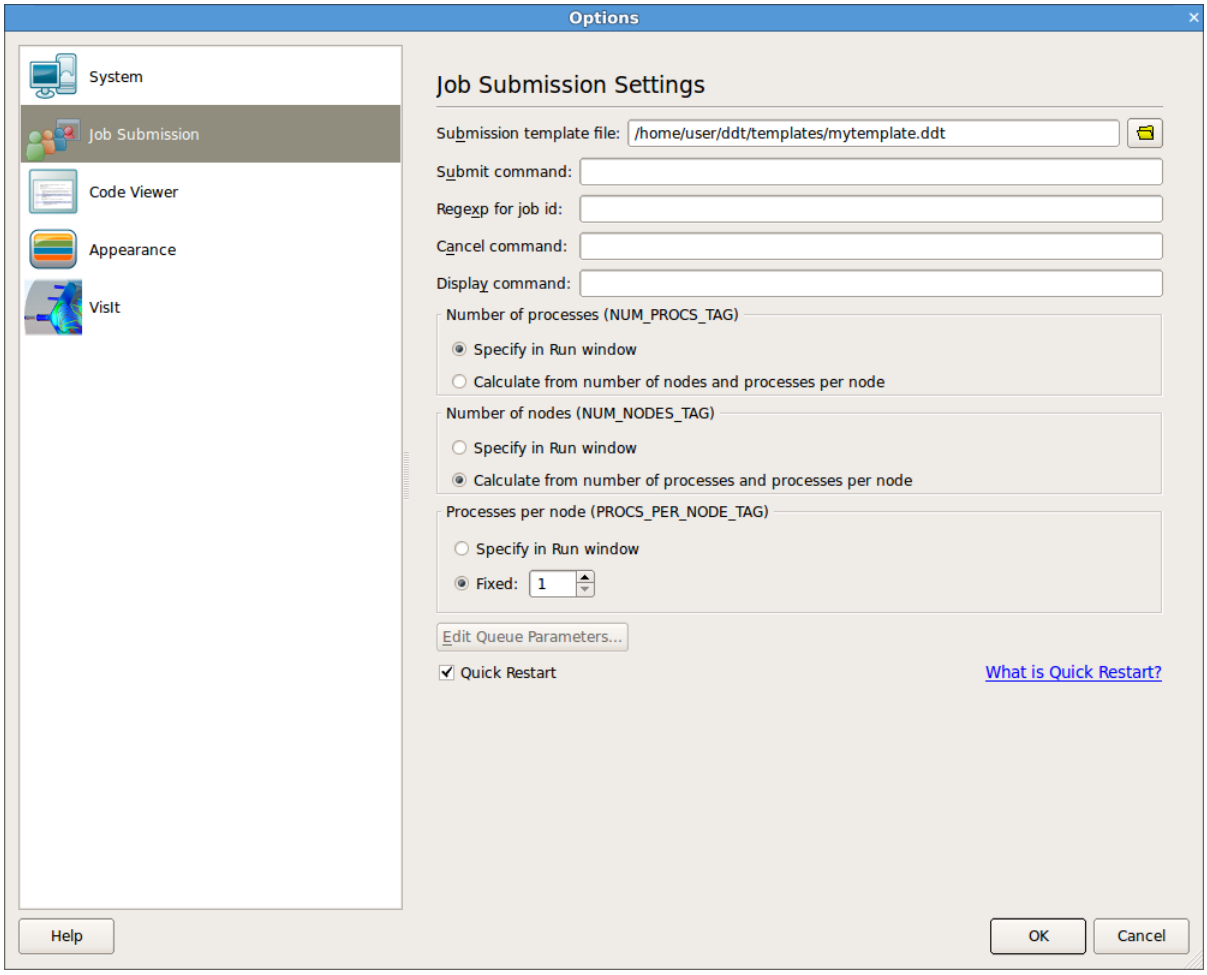


Figure 21: *DDT Using Substitute MPI Commands*

Note the *Submit Command* and the *Submission Template File* in particular. DDT will create a new file and append it to the submit command before executing it. So, in this case what would actually be executed might be `mpiexec-config /tmp/ddt-temp-0112` or similar. Therefore, any argument like `-config` must be last on the line, because DDT will add a file name to the end of the line. Other arguments, if there are any, can come first.

We recommend reading the section on queue submission, as there are many features described there that might be useful to you if your system uses a non-standard start up command. If you do use a non-standard command, please email us at support@allinea.com and let us know about it – you might find the next version supports it out-of-the-box!

4.11 Starting DDT From A Job Script

The usual way of debugging a program with DDT in a queue/batch environment is to configure DDT to submit the program to the queue for you (See section 17.6 Starting A Job In A Queue above).

© 2015 Allinea Software Ltd.

Some users may wish to start DDT itself from a job script that is submitted to the queue/batch environment. To do this:

1. Configure DDT with the correct MPI implementation.
2. Disable queue submission in the DDT options.
3. Create a job script that starts DDT using the command:

   ```
   ddt -start -noqueue -once -n NPROCS -- PROGRAM [ARGUMENTS]...
   ```

   where `NPROCS` is the number of processes to start `PROGRAM` is the program to run and `ARGUMENTS` are the arguments to the program.
4. Submit the job script to the queue. The `-once` argument tells DDT to exit when the session ends.

© 2015 Allinea Software Ltd.

5 DDT: Overview

DDT uses a tabbed-document interface – a method of presenting multiple documents that is familiar from many present day applications. This allows you to have many source files open, and to view one (or two, if the *Source Code Viewer* is 'split') in the full workspace area.

Each component of DDT (labelled and described in the key) is a dockable window, which may be dragged around by a handle (usually on the top or left-hand edge). Components can also be double-clicked, or dragged outside of DDT, to form a new window. You can hide or show most of the components using the *View* menu. The screen shot shows the default DDT layout.

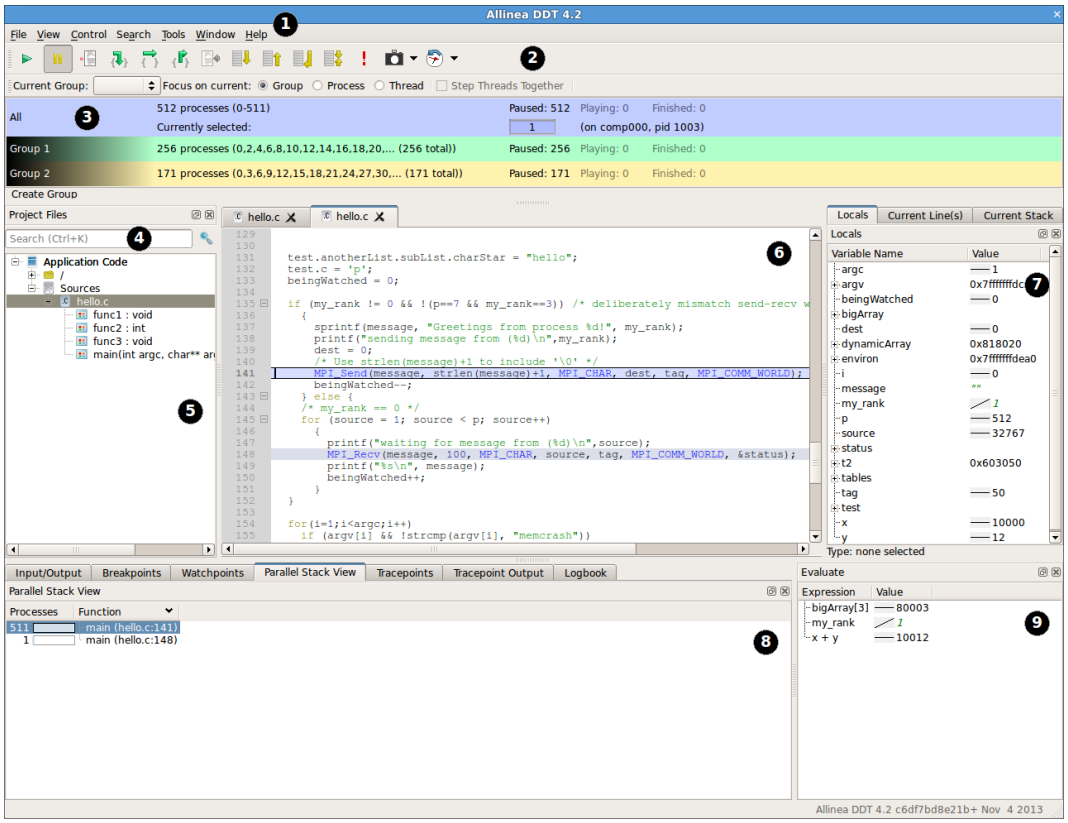


Figure 22: *DDT Main Window*

Key
(1) Menu Bar
(2) Process Controls
(3) Process Groups
(4) Find File or Function
(5) Project Files
(6) Source Code
(7) Variables and Stack of Current Process/Thread
(8) Parallel Stack, IO and Breakpoints
(9) Evaluate Window
(10) Status Bar

Please note that on some platforms, the default screen size can be insufficient to display the status bar –

if this occurs, you should expand the DDT window until DDT is completely visible.

5.1 Saving And Loading Sessions

Most of the user-modified parameters and windows are saved by right-clicking and selecting a save option in the corresponding window.

However, DDT also has the ability to load and save all these options concurrently to minimize the inconvenience in restarting sessions. Saving the session stores such things as *Process Groups*, the contents of the *Evaluate* window and more. This ability makes it easy to debug code with the same parameters set time and time again.

To save a session simply use the *Save Session* option from the *File* menu. Enter a file name (or select an existing file) for the save file and click OK. To load a session again simply choose the *Load Session* option from the *File* menu, choose the correct file and click OK.

5.2 Source Code

When DDT begins a session, source code is automatically found from the information compiled in the executable.

Source and header files found in the executable are reconciled with the files present on the front-end server, and displayed in a simple tree view within the *Project Files* tab of the *Project Navigator* window. Source files can be loaded for viewing by clicking on the file name.

Whenever a selected process is stopped, the *Source Code Viewer* will automatically leap to the correct file and line, if the source is available.

The source code viewer supports automatic colour syntax highlighting for C and Fortran.

You can hide functions or subroutines you are not interested in by clicking the - glyph next to the first line of the function. This will collapse the function. Simply click the + glyph to expand the function again.

5.3 Project Files

The *Project Files* tree shows a list of source files for your program. Click on a file in the tree to open it in the *Code Viewer*. You may also expand a source file to see a list of functions / procedures defined in that source file (C / C++ / Fortran only).

5.3.1 Application / External Code

DDT automatically splits your source code into *Application Code* (source code from your application itself) and *External Code* (code from third party libraries). This allows you to quickly distinguish between your own code and, for example, third party libraries.

You can control exactly which directories are considered to contain Application Code using the *Application / External Directories* window. Right-click on the *Project Files* tree to open the window.

The checked directories are the directories containing Application Code. Once you have configured them to your satisfaction click *Ok* to update the *Project Files* tree.

© 2015 Allinea Software Ltd.

5.4 Finding Lost Source Files

On some platforms, not all source files are found automatically. This can also occur, for example, if the executable or source files have been moved since compilation. Extra directories to search for source files can be added by right-clicking whilst in the *Project Files* tab, and selecting *Add/view Source Directory(s)*. You can also specify extra source directories on the command line using the `-source-dirs` command line argument (separate each directory with a colon).

It is also possible to add an individual file – if, for example, this file has moved since compilation or is on a different (but visible) file system – by right-clicking in the *Project Files* tab and selecting the *Add File* option.

Any directories or files you have added are saved and restored when you use the *Save Session* and *Load Session* commands inside the *File* menu. If DDT doesn't find the sources for your project, you might find these commands save you a lot of unnecessary clicking.

5.5 Finding Code Or Variables

5.5.1 Find File or Function

The *Find File Or Function Box* appears above the source file tree. You can type the name of a file or function in this box to search for that file/function in the source file tree. You can also type just part of a name to see all the files/functions whose name contains the text you typed. Double-click on a result to jump to the corresponding line for that file/function.

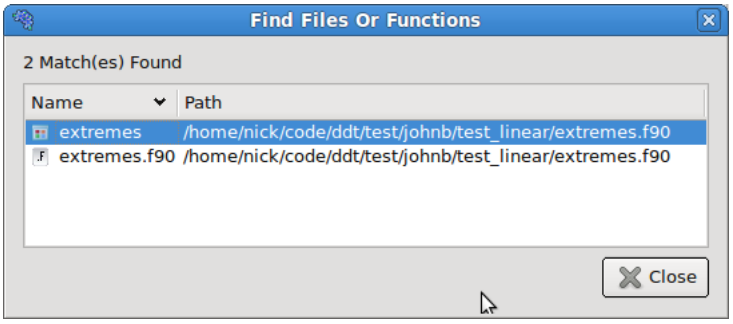


Figure 23: *Find Files or Function box*

5.5.2 Find

The *Find* menu item can be found in the *Search* menu, and can be used to find occurrences of an expression in the currently visible source file.

DDT will search from the current cursor position for the next or previous occurrence of the search term. Click on the magnifying glass icon for more search options.

Case Sensitive: When checked, DDT will perform a case sensitive search (e.g. `Hello` will not match `hello`).

Whole Words Only: When checked, DDT will only match your search term against whole 'words' in the source file. For example `Hello` would not match `HelloWorld` while searching for whole words only.

Use Regular Expressions: When this is checked, your search may use Perl style regular expressions.

© 2015 Allinea Software Ltd.

5.5.3 Find in Files

The *Find In Files* window can be found in the *Search* menu, and can be used to search all source and header files associated with your program. The search results are listed and can be clicked to display the file and line number in the main *Source Code Viewer*; this can be of particular use for setting a breakpoint at a function.

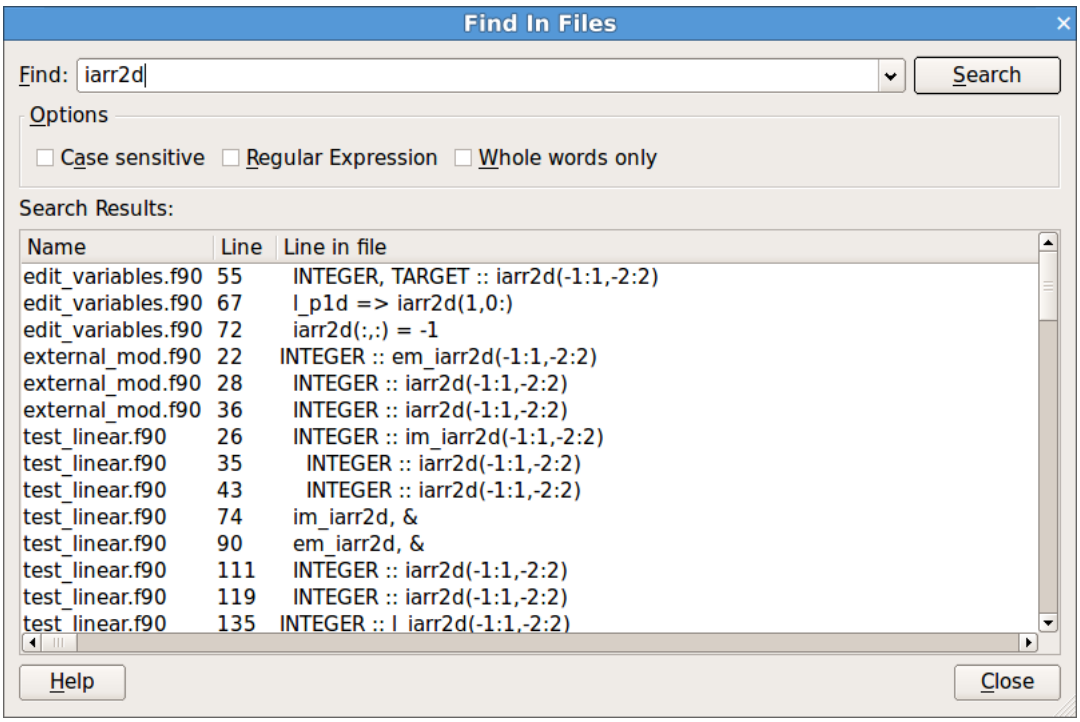


Figure 24: *Find in Files dialog*

Case sensitive: When checked, DDT will perform a case sensitive search (e.g. `Hello` will not match `hello`).

Whole words only: When checked, DDT will only match your search term against whole 'words' in the source file. For example `Hello` would not match `HelloWorld` while searching for whole words only.

Regular Expression: When checked, DDT will interpret the search term as a regular expression rather than a fixed string. The syntax of the regular expression is identical to that described in the appendix F.6 Job ID Regular Expression.

5.6 Jump To Line / Jump To Function

DDT has a jump to line function which enables the user to go directly to a line of code. This is found in the *Search* menu. A window will appear in the centre of your screen. Enter the line number you wish to see and click *OK*. This will take you to the correct line providing that you entered a line that exists. You can use the hotkey *CTRL-G* to access this function quickly.

DDT also allows you to jump directly to the implementation of a function. In the *Project Files* tab of the *Project Navigator* window on the left side of the main screen you should see small + symbols next to each file:

© 2015 Allinea Software Ltd.

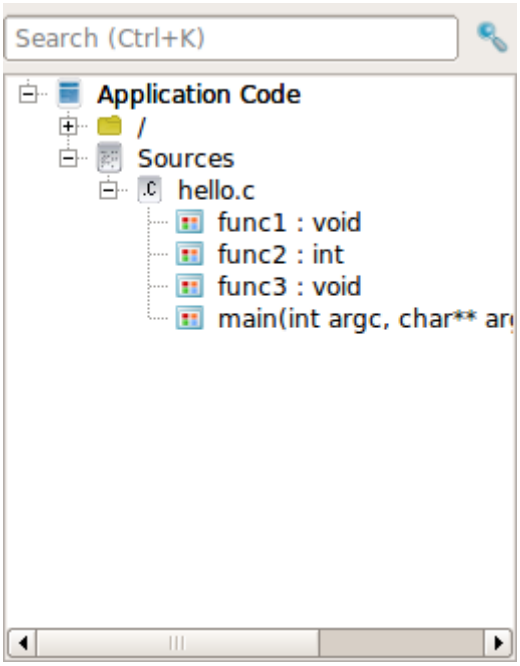


Figure 25: *Function Listing*

Clicking on a the + will display a list of the functions in that file. Clicking on any function will display it in the Source Code viewer.

5.7 Static Analysis

Static analysis is a powerful companion to debugging. Whilst Allinea DDT enables the user to discover errors by code and state inspection along with automatic error detection components such as memory debugging, static analysis inspects the source code and attempts to identify errors that can be detected from the source alone – independently of the compiler and actual process state.

Allinea DDT includes the static analysis tools `cppcheck` and `ftnchek`. These will by default automatically examines source files as they are loaded and display a warning symbol if errors are detected. Typical errors include:

- Buffer overflows – accessing beyond the bounds of heap or stack arrays
- Memory leaks – allocating memory within a function and there being a path through the function which does not deallocate the memory and the pointer is not assigned to any externally visible variable, nor returned.
- Unused variables, and also use of variables without initialization in some cases.

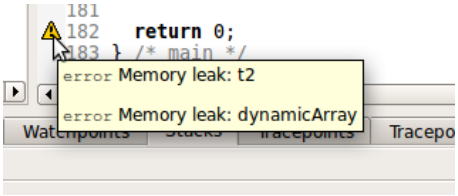


Figure 26: *Static Analysis Error Annotation*

Static analysis is not guaranteed to detect all, or any, errors, and an absence of warning triangles should

© 2015 Allinea Software Ltd.

not be considered to be an absence of bugs.

5.8 Editing Source Code

You can the right click in the Source Code Viewer and select the Open file in editor option to open the current file in the default editor for your desktop environment. If you want to change the editor used, or the file doesn't open with the default settings, open the Options window by selecting *File* → *Options* (*DDT* → *Preferences* on Mac OS X) and enter the path of your preferred editor in the Editor box, e.g. `/usr/bin/gedit`.

Note: Editing source code is not possible when using the remote client.

5.9 Version Control Information

The version control integration in DDT and MAP allows users to see line-by-line information from Git, Mercurial or Subversion next to source files. Information is colour-coordinated to indicate the age of the source line.

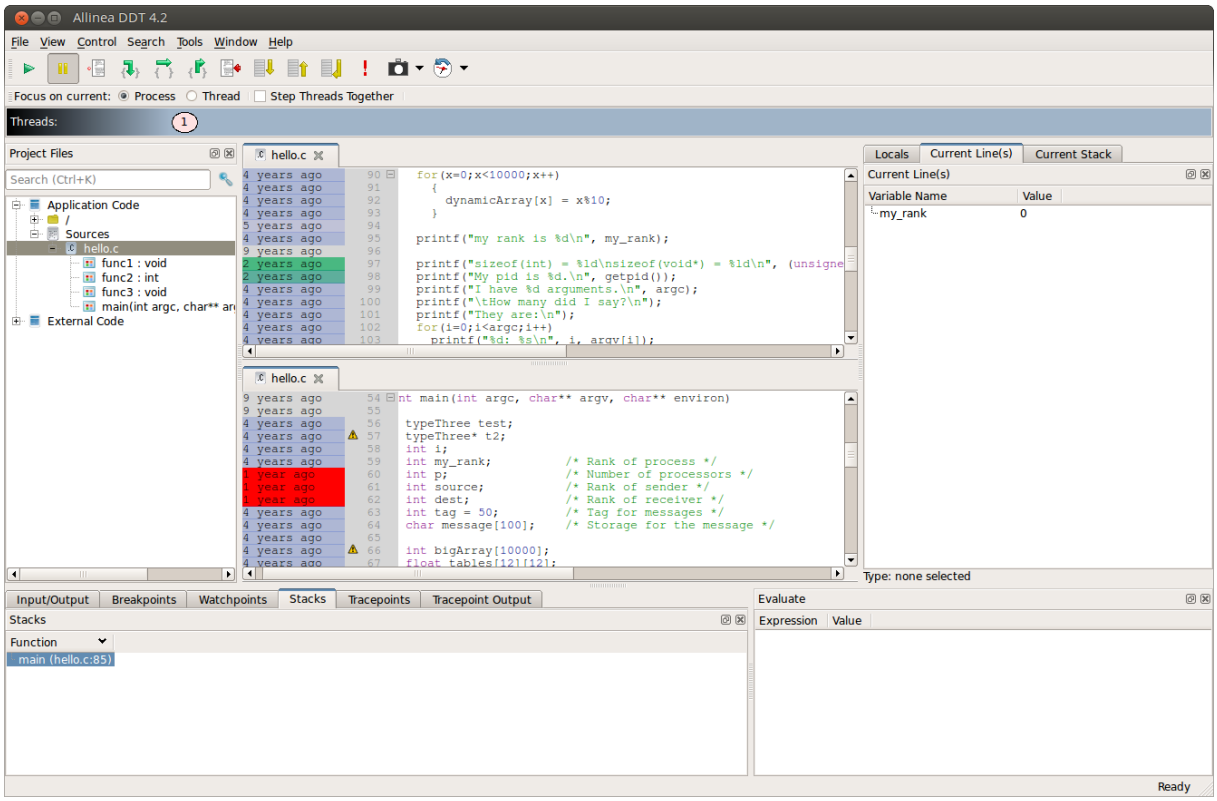


Figure 27: *DDT running with Version Control Information enabled*

To enable select the *Version Control Information* option from the *View* menu. When enabled columns to left of source code viewers are shown. In these columns are displayed how long ago the line was added/modified. Each line in the information column is highlighted in a colour to indicate its age. The lines changed in the current revision are highlighted in red. Where available lines with changes not committed are highlighted in purple. All other lines are highlighted with a blend of transparent blue and opaque green where blue indicates old and green young. Currently uncommitted changes are only supported for Git. DDT/MAP will not show ANY version control information for files with uncommitted changes when using Mercurial or Subversion.

© 2015 Allinea Software Ltd.

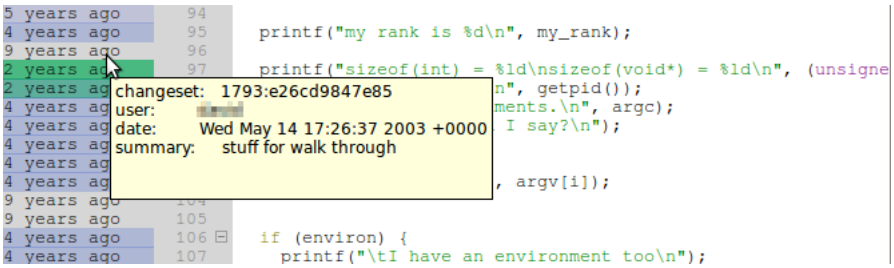

changeset: 1793:e26cd9847e85
user:
date: Wed May 14 17:26:37 2003 +0000
summary: stuff for walk through
printf("my rank is %d\n", my_rank);
if (environ) {
printf("\tI have an environment too\n");

Figure 28: *Version Control Information - Tooltips*

A folded block of code displays the annotation for the most recently modified line in the block.

Hovering the cursor over the information column reveals a tool-tip containing a preview of the commit message for the commit that last changed the line.

Figure 29: *Version Control Information - Context Menu*

To copy the commit message right-click the column on the desired row and from the menu select Copy Commit Message.

© 2015 Allinea Software Ltd.

6 DDT: Controlling Program Execution

Whether debugging a multi-process or a single process code, the mechanisms for controlling program execution are very similar.

In multi-process mode, most of the features described in this section are applied using *Process Groups*, which we describe now. For single process mode, the commands and behaviours are identical, but apply to only a single process – freeing the user from concerns about process groups.

6.1 Process Control And Process Groups

MPI programs are designed to run as more than one process and can span many machines. DDT allows you to group these processes so that actions can be performed on more than one process at a time. The status of processes can be seen at a glance by looking at the *Process Group Viewer*.

The Process Group Viewer is (by default) at the top of the screen with multi-coloured rows. Each row relates to a group of processes and operations can be performed on the currently highlighted group (e.g. playing, pausing and stepping) by clicking on the toolbar buttons. Switch between groups by clicking on them or their processes - the highlighted group is indicated by a lighter shade. Groups can be created, deleted, or modified by the user at any time, with the exception of the *All* group, which cannot be modified.

Groups are added by clicking on the *Create Group* button or from a context-sensitive menu that appears when you right-click on the process group widget. This menu can also be used to rename groups, delete individual processes from a group and jump to the current position of a process in the code viewer. You can load and save the current groups to a file, and you can create sub-groups from the processes currently playing, paused or finished. You can even create a sub-group excluding the members of another group – for example, to take the complement of the *Workers* group, select the *All* group and choose *Copy, but without Workers*.

You can also use the context menu to switch between the two different ways of viewing the list of groups in DDT – the detailed view and the summary view:

6.1.1 Detailed View

The detailed view is ideal for working with smaller numbers of processes. If your program has less than 32 processes, DDT will default to the detailed view. You can switch to this view using the context menu if you wish.

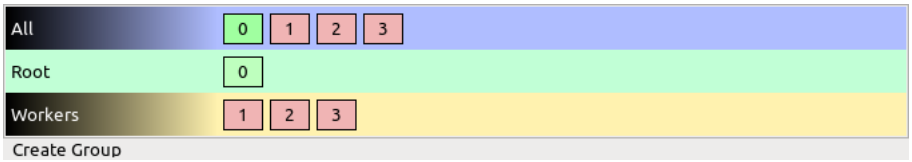


Figure 30: *The Detailed Process Group View*

In the detailed view, each process is represented by a square containing its MPI rank (0 through n-1). The squares are colour-coded; red for a paused process, green for a playing process and grey for a finished/dead process. Selected processes are highlighted with a lighter shade of their colour and the current process also has a dashed border.

© 2015 Allinea Software Ltd.

When a single process is selected the local variables are displayed in the *Variable Viewer* and displayed expressions are evaluated. You can make the *Source Code Viewer* jump to the file and line for the current stack frame (if available) by double-clicking on a process.

To copy processes from one group to another, simply click and drag the processes. To delete a process, press the delete key. When modifying groups it is useful to select more than one process by holding down one or more of the following:

Key	Description
Control	Click to add/remove process from selection
Shift	Click to select a range of processes
Alt	Click to select an area of processes

Note: Some window managers (such as KDE) use Alt and drag to move a window - you must disable this feature in your window manager if you wish to use the DDT's area select.

6.1.2 Summary View

The summary view is ideal for working with moderate to huge numbers of processes. If your program has 32 processes or more, DDT will default to this view. You can switch to this view using the context menu if you wish.

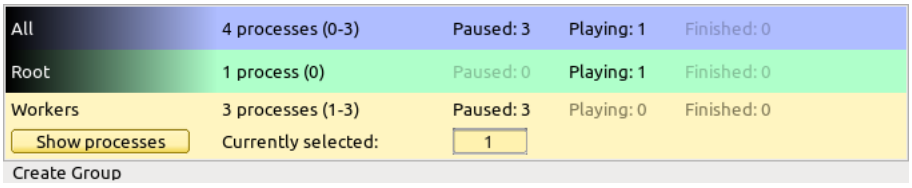


Figure 31: *The Summary Process Group View*

In the summary view, individual processes are not shown. Instead, for each group, DDT shows:

- The number of processes in the group.
- The processes belonging that group – here *1-2048* means processes 1 through 2048 inclusive, and *1-10, 12-1024* means processes 1-10 and processes 12-1024 (but not process 11). If this list becomes too long, it will be truncated with a '...'. Hovering the mouse over the list will show more details.
- The number of processes in each state (playing, paused or finished). Hovering the mouse over each state will show a list of the processes currently in that state.
- The rank of the currently selected process. You can change the current process by clicking here, typing a new rank and pressing Enter. Only ranks belonging to the current group will be accepted.

The *Show processes* toggle button allows you to switch a single group into the detailed view and back again – handy if you're debugging a 2048 process program but have narrowed the problem down to just 12 processes, which you've put in a group.

© 2015 Allinea Software Ltd.

6.2 Focus Control

Figure 32: *Focus options*

The focus control allows you to focus on individual processes or threads as well as process groups. When focused on a particular process or thread, actions such as stepping, playing/pausing, adding breakpoints etc. will only apply to that process/thread rather than the entire group. In addition, the DDT GUI will change depending on whether you're focused on group, process or thread. This allows DDT to display more relevant information about your currently focused object.

6.2.1 Overview of changing focus

Focusing in DDT affects a number of different controls in the DDT main window. These are briefly described below:

Note: Focus controls do not affect DDT windows such as the Multi Dimensional Array Viewer, Memory Debugger, Cross Process Comparison etc.

6.2.2 Process Group Viewer

The changes to the process group viewer amongst the most obvious changes to the DDT GUI. When focus on current group is selected you will see your currently created process groups. When switching to focus on current process or thread you will see the view change to show the processes in the currently selected group, with their corresponding threads.

Figure 33: *The Detailed Process Group View Focused on a Process*

If there are 32 threads or more, DDT will default to showing the threads using a summary view (as in the Process Group View). The view mode can also be changed using the context menu.

During focus on process, a tooltip will be shown that identifies the OpenMP thread ID of each thread, if the value exists.

6.2.3 Breakpoints

The breakpoints tab in DDT will be filtered to only display breakpoints relevant to your current group, process, thread. When focused on a process, The breakpoint tab will display which thread the breakpoint belongs to. If you are focused on a group, the tab will display both the process and the thread the breakpoint belongs to.

© 2015 Allinea Software Ltd.

6.2.4 Code Viewer

The code viewer in DDT shows a stack back trace of where each thread is in the call stack. This will also be filtered by the currently focused item, for example when focused on a particular process, you will only see the back trace for the threads in that process.

Also, when adding breakpoints using the code viewer, they will be added for the group, process or thread that is currently focused.

6.2.5 Parallel Stack View

The parallel stack view can also be filtered by focusing on a particular process group, process or thread.

6.2.6 Playing and Stepping

The behaviour of playing, stepping and the *Run* to here feature are also affected by your currently focused item. When focused on a process group, the entire group will be affected, whereas focusing on a thread will mean that only current thread will be executed. The same goes for processes, but with an additional option which is explained below.

6.2.7 Step Threads Together

The step threads together feature in DDT is only available when focused on process. If this option is enabled then DDT will attempt to synchronise the threads in the current process when performing actions such as stepping, pausing and using *Run to here*.

For example, if you have a process with 2 threads and you choose *Run to here*, DDT will pause your program when either of the threads reaches the specified line. If *Step threads together* is selected DDT will attempt to play both of the threads to the specified line before pausing the program.

Important note: You should always use *Step threads together* and Run to here to enter or move within OpenMP parallel regions. With many compilers it is also advisable to use *Step threads together* when leaving a parallel region, otherwise threads can get 'left behind' inside system-specific locking libraries and may not enter the next parallel region on the first attempt.

6.2.8 Stepping Threads Window

When using the step threads together feature it is not always possible for all threads to synchronise at their target. There are two main reasons for this:

1. One or more threads may branch into a different section of code (and hence never reach the target). This is especially common in OpenMP codes, where worker threads are created and remain in holding functions during sequential regions.
2. As most of DDT's supported debug interfaces cannot play arbitrary groups of threads together, DDT simulates this behaviour by playing each thread in turn. This is usually not a problem, but can be if, for example, thread 1 is playing, but waiting for thread 2 (which is not currently playing). DDT will attempt to resolve this automatically but cannot always do so.

If either of these conditions occur, the Stepping Threads Window will appear, displaying the threads which have not yet reached their target.

© 2015 Allinea Software Ltd.

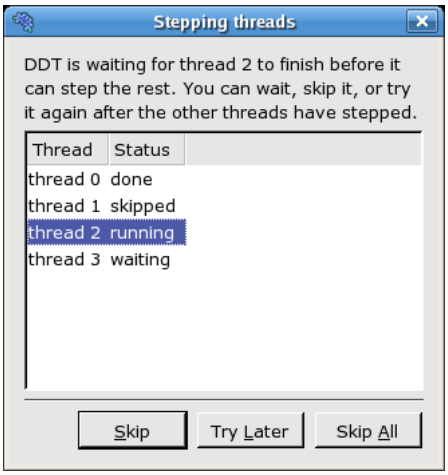

Stepping threads
DDT is waiting for thread 2 to finish before it can step the rest. You can wait, skip it, or try it again after the other threads have stepped.
Thread
Status
thread 0 done
thread 1 skipped
thread 2 running
thread 3 waiting
Skip
Try Later
Skip All

Figure 34: *The Stepping Threads Window*

The stepping threads window also displays the status of threads, which may be one of the following:

- **Done:** The thread has reached it target (and has been paused).
- **Skipped:** The thread has been skipped (and paused). DDT will no longer wait for it to reach it's target.
- **Playing:** This is the thread that is currently being executed. Only one thread may be playing at a time while the Stepping Threads Window is open.
- **Waiting:** The thread is currently awaiting execution. When the currently *playing* thread is *done* or has been skipped, the highest *waiting* thread in the list will be executed.

The stepping threads window also lets you interact with the threads with the following options:

- **Skip:** DDT will skip and pause the currently playing thread. If this is the last waiting thread the window will be closed.
- **Try Later:** The currently playing thread will be paused, and added to the bottom of the list of threads to be retried later. This is useful if you have threads which are waiting on each other.
- **Skip All:** This will skip (and pause) all of the threads and close the window.

© 2015 Allinea Software Ltd.

6.3 Hotkeys

DDT comes with a pre-defined set of hotkeys to enable easy control of your debugging. All the features you see on the toolbar and several of the more popular functions from the menus have hotkeys assigned to them. Using the hotkeys will speed up day to day use of DDT and it is a good idea to try to memorize these.

Key	Function
F9	Play
F10	Pause
F5	Step into
F8	Step over
F6	Step out
CTRL-D	Down stack frame
CTRL-U	Up stack frame
CTRL-B	Bottom stack frame
CTRL-A	Align stack frames with current
CTRL-G	Go to line number
CTRL-F	Find

6.4 Starting, Stopping and Restarting a Program

The *File* menu can be accessed at almost any time while DDT is running. If a program is running you can end it and run it again or run another program. When DDT's start up process is complete your program should automatically stop either at the main function for non-MPI codes, or at the `MPI_Init` function for MPI.

When a job has run to the end DDT will show a window box asking if you wish to restart the job. If you select yes then DDT will kill any remaining processes and clear up the temporary files and then restart the session from scratch with the same program settings.

When ending a job, DDT will attempt to ensure that all the processes are shut down and clear up any temporary files. If this fails for any reason you may have to manually `kill` your processes using kill, or a method provided by your MPI implementation such as `lamclean` for LAM/MPI.

6.5 Stepping Through A Program

To continue the program playing click *Play/Continue* and to stop it at any time click *Pause* .For multi-process DDT these start/stop all the processes in the current group (see *Process Control* and *Process Groups*).

Like many other debuggers there are three different types of step available. The first is *Step Into* that will move to the next line of source code unless there is a function call in which case it will step to the first line of that function. The second is *Step Over* that moves to the next line of source code in the bottom stack frame. Finally, *Step Out* will execute the rest of the function and then stop on the next line in the stack frame above. The return value of the function is displayed in the *Locals* view. When using *Step Out* be careful not to try and step out of the main function, as doing this will end your program.

© 2015 Allinea Software Ltd.

6.6 Stop Messages

In certain circumstances your program may be automatically paused by the debugger. There are five reasons your program may be paused in this way:

1. It hit one of DDT's default breakpoints (e.g. `exit` or `abort`). See section 6.11 Default Breakpoints for more information on default breakpoints.
2. It hit a user-defined breakpoint (a breakpoint shown in the *Breakpoints* view).
3. The value of a watched variable changed.
4. It was sent a signal. See section 6.21 Signal Handling for more information on signals.
5. It encountered a Memory Debugging error. See section 11.4 Pointer Error Detection and Validity Checking for more information on Memory Debugging errors.

DDT will display a message telling you exactly why the program was paused. The text may be copied to the clipboard by selecting it with the mouse, then right-clicking and selecting *Copy*. You may want to suppress these messages in certain circumstances, for example if you are playing from one breakpoint to another. Use the *Control* → *Messages* menu to enable/disable stop messages.

6.7 Setting Breakpoints

6.7.1 Using the Source Code Viewer

First locate the position in your code that you want to place a breakpoint at. If you have a lot of source code and wish to search for a particular function you can use the *Find/Find In Files* window. Clicking the right mouse button in the *Source Code Viewer* displays a menu showing several options, including one to add or remove a breakpoint. In multi-process mode this will set the breakpoint for every member of the current group. Breakpoints may also be added by left clicking margin to the left of the line number.

Every breakpoint is listed under the breakpoints tab towards the bottom of DDT's window.

If you add a breakpoint at a location where there is no executable code, DDT will highlight the line you selected as having a breakpoint. However when hitting the breakpoint, DDT will stop at the next executable line of code.

6.7.2 Using the Add Breakpoint Window

You can also add a breakpoint by clicking the *Add Breakpoint* icon in the toolbar. This will open the *Add Breakpoint* window.

© 2015 Allinea Software Ltd.

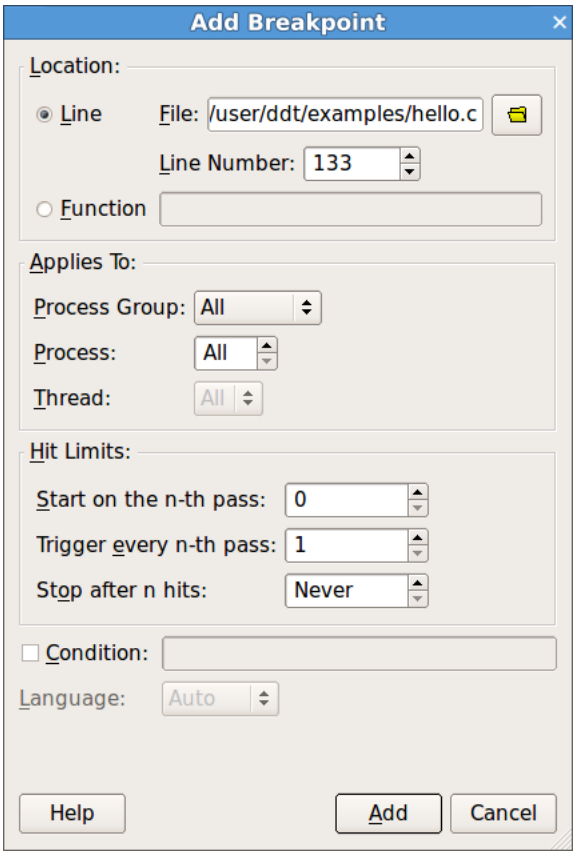


Figure 35: *The Add Breakpoint window*

You may wish to add a breakpoint in a function for which you do not have any source code: for example in `malloc`, `exit`, or `printf` from the standard system libraries. Select the *Function* radio button and enter the name of the function in the box next to it.

You can specify what group/process/thread you want the breakpoint to apply in the *Applies To* section. You may also make the breakpoint conditional by checking the *Condition* check box and entering a condition in the box.

6.7.3 Pending Breakpoints

Note: This feature is not supported on all platforms.

If you try to add a breakpoint on a function that is not defined, DDT will ask if you want to add it anyway. If you click *Yes* the breakpoint will be applied to any shared objects that are loaded in the future.

© 2015 Allinea Software Ltd.

6.7.4 Conditional Breakpoints

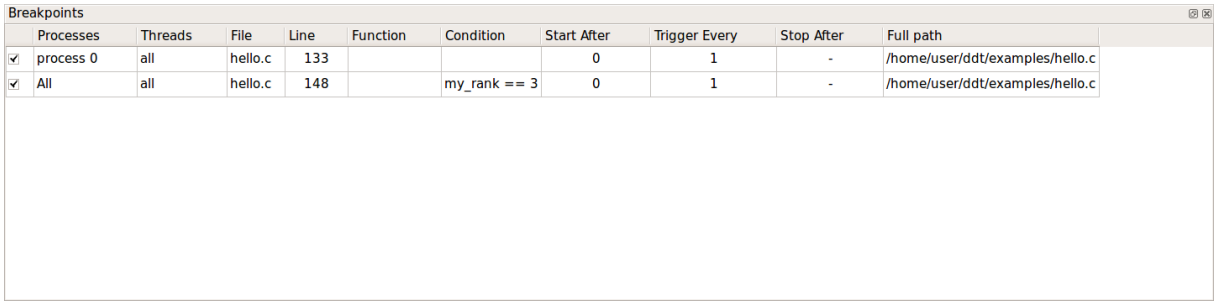

Breakpoints
Processes Threads File Line Function Condition Start After Trigger Every Stop After Full path
process 0 all hello.c 133 0 1 - /home/user/ddt/examples/hello.c
All all hello.c 148 my_rank == 3 0 1 - /home/user/ddt/examples/hello.c

Figure 36: *The Breakpoints Table*

Select the breakpoints tab to view all the breakpoints in your program. You may add a condition to any of them by clicking on the condition cell in the breakpoint table and entering an expression that evaluates to *true* or *false*. Each time a process (in the group the breakpoint is set for) passes this breakpoint it will evaluate the condition and break only if it returns *true* (typically any non-zero value). You can drag an expression from the *Evaluate* window into the condition cell for the breakpoint and this will be set as the condition automatically.

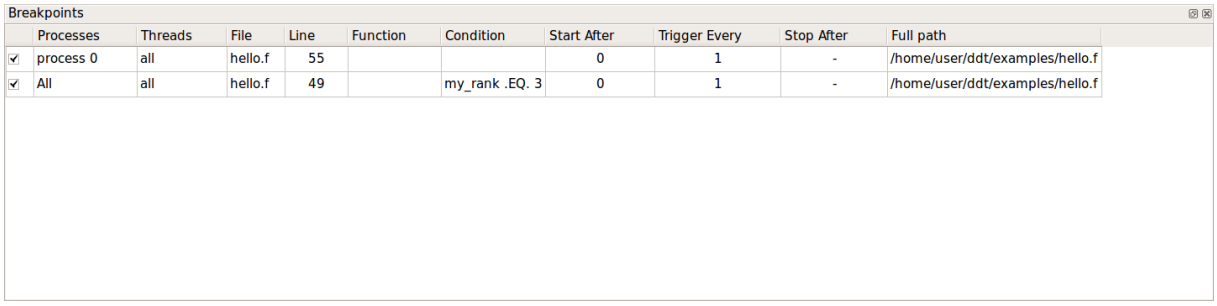

Breakpoints
Processes Threads File Line Function Condition Start After Trigger Every Stop After Full path
process 0 all hello.f 55 0 1 - /home/user/ddt/examples/hello.f
All all hello.f 49 my_rank .EQ. 3 0 1 - /home/user/ddt/examples/hello.f

Figure 37: *Conditional Breakpoints In Fortran*

The expression should be in the same language as your program. Also, please note the condition evaluator is quite pedantic with Fortran conditions, and to ensure the correct interpretation of compound boolean operations, it is advisable to bracket your expressions amply.

6.8 Suspending Breakpoints

A breakpoint can be temporarily deactivated and reactivated by checking/unchecking the activated column in the breakpoints panel.

6.9 Deleting A Breakpoint

Breakpoints may be deleted by right-clicking on the breakpoint in the breakpoints panel, or by right-clicking at the file/line of the breakpoint whilst in the correct process group and right-clicking and selecting delete breakpoint. They may also be deleted by left clicking the breakpoint icon in the margin to the left of the line number in the code viewer.

© 2015 Allinea Software Ltd.

6.10 Loading And Saving Breakpoints

To load or save the breakpoints in a session right-click in the breakpoint panel and select the load/save option. Breakpoints will also be loaded and saved as part of the load/save session.

6.11 Default Breakpoints

DDT has a number of default breakpoints that will stop your program under certain conditions which are described below. You may enable/disable these while your program is running using the *Control* → *Default* Breakpoints menu.

- *Stop at exit/_exit*

  When enabled, DDT will pause your program as it is about to end under normal exit conditions. DDT will pause both before and after any exit handlers have been executed. (Disabled by default.)

- *Stop at abort/fatal MPI Error*

  When enabled, DDT will pause your program as it about to end after an error has been triggered. This includes MPI and non-MPI errors. (Enabled by default.)

- *Stop on throw (C++ exceptions)*

  When enabled, DDT will pause your program whenever an exception is thrown (regardless of whether or not it will be caught). Due to the nature of C++ exception handling, you may not be able to step your program properly at this point. Instead, you should play your program or use the Run to here feature in DDT. (Disabled by default.)

- *Stop on catch (C++ exceptions)*

  As above, but triggered when your program catches a thrown exception. Again, you may have trouble stepping your program. (Disabled by default.)

- *Stop at fork*

  DDT will stop whenever your program forks (i.e. calls the fork system call to create a copy of the current process). The new process is added to your existing DDT session and can be debugged along with the original process.

- *Stop at exec*

  When your program calls the exec system call, DDT will stop at the main function (or program body for Fortran) of the new executable.

- *Stop on CUDA kernel launch*

  When debugging CUDA GPU code, this will pause your program at the entry point of each kernel launch.

- *Stop on Xeon Phi offload*

  Stops your program when an offload process is started and attaches to the offload process. You can then set breakpoints in offloaded code.

6.12 Synchronizing Processes

If the processes in a process group are stopped at different points in the code and you wish to re-synchronize them to a particular line of code this can be done by right-clicking on the line at which you wish to syn-

chronize them to and selecting *Run To Here*. This effectively plays all the processes in the selected group and puts a break point at the line at which you choose to synchronize the processes at, ignoring any breakpoints that the processes may encounter before they have synchronized at the specified line.

If you choose to synchronize your code at a point where all processes do not reach then the processes that cannot get to this point will play to the end.

Note: Though this ignores breakpoints while synchronizing the groups it will not actually remove the breakpoints.

Note: If a process is already at the line which you choose to synchronize at, the process will still be set to play. Be sure that your process will revisit the line, or alternatively synchronize to the line immediately after the current line.

6.13 Setting A Watchpoint

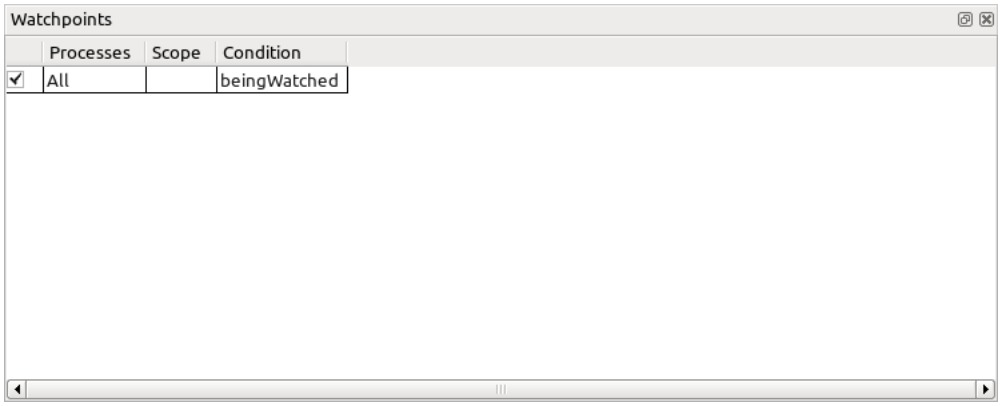


Figure 38: *The Watchpoints Table*

You can set a watchpoint on a variable or expression that causes DDT to stop every time it changes.

Figure 39: *Program Stopped At Watchpoint being watched*

Unlike breakpoints, watchpoints are not displayed in the *Source Code Viewer*. Instead they are created by right-clicking on the *Watchpoints* view and selecting the *Add Watchpoint* menu item, or dragging a variable to the *Watchpoints* view from the *Local Variables*, *Current Line* or *Evaluate* views.

Upon adding a watchpoint the *Add Watchpoint* dialog appears allowing you to apply restrictions to the watchpoint. *Process Group* restricts the watch point to the chosen process group (see 6.1 Process Control And Process Groups). *Process* restricts the watchpoint to the chosen process. *Expression* is the variable name in the program to be watched. *Language* is the language of the portion of the program containing the expression.

© 2015 Allinea Software Ltd.

You can set a watchpoint for either a single process, or every process in a process group.

DDT will automatically remove a watchpoint once the target variable goes out of scope. If you are watching the value pointed to by a variable, i.e. `*p` - you may want to continue watching the value at that address even after `p` goes out of scope. You can do this by right-clicking on `*p` in the Watchpoints view and selecting the *Pin to address* menu item. This replaces the variable `p` with its address so the watch will not be removed when `p` goes out of scope.

6.14 Tracepoints

Tracepoints allow you to see what lines of code your program is executing – and the variables – without stopping it. Whenever a thread reaches a tracepoint it will print the file and line number of the tracepoint to the Input/Output view. You can also capture the value of any number of variables or expressions at that point.

Examples of situations in which this feature will prove invaluable include

- Recording entry values in a function that is called many times, but crashes only occasionally. Setting a tracepoint makes it easier to correlate the circumstances that cause a crash.
- Recording entry to multiple functions in a library – enabling the user (or library developer) to check which functions are being called, and in which order. An example of this is the MPI History Plugin (see Section 13.3 Using a Plugin, Section of this guide) which records MPI usage.
- Observing progress of an application and variation of values across processes without having to interrupt the application.

6.14.1 Setting a Tracepoint

Tracepoints are added by either right-clicking on a line in the *Source Code Viewer* and selecting the *Add Tracepoint* menu item, or by right-clicking in the *Tracepoints* view and selecting *Add Tracepoint*. If you right-click in the *Source Code Viewer* a number of variables based on the current line of code will be captures by default.

Tracepoints can lead to considerable resource consumption by the user interface if placed in areas likely to generate a lot of passing. For example, if a tracepoint is placed inside of a loop with N iterations, then N separate tracepoint passings will be recorded. Whilst Allinea DDT will attempt to merge such data scalably, when alike tracepoints are passed in order between processes, where process behaviour is likely to be divergent and unmergeable then a considerable load would then be caused.

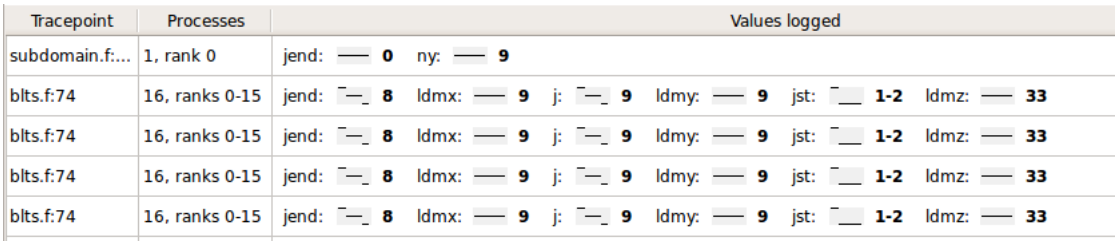
If it is necessary to place a tracepoint inside a loop, set a condition on the tracepoint to ensure you only log what is of use to you.

Tracepoints also momentarily stop processes at the tracepoint location in order to evaluate the expressions and record their values, and hence if placed inside (eg.) a loop with a very large number of iterations – or a function executed many times per second, then a slow down in the pace of your application will be noticed.

6.14.2 Tracepoint Output

The output from the tracepoints can be found in the *Tracepoint Output* view.

© 2015 Allinea Software Ltd.



Tracepoint	Processes	Values logged
subdomain.f:...	1, rank 0	jend: 0 ny: 9
blts.f:74	16, ranks 0-15	jend: 8 ldmx: 9 j: 9 ldmy: 9 jst: 1-2 ldmz: 33
blts.f:74	16, ranks 0-15	jend: 8 ldmx: 9 j: 9 ldmy: 9 jst: 1-2 ldmz: 33
blts.f:74	16, ranks 0-15	jend: 8 ldmx: 9 j: 9 ldmy: 9 jst: 1-2 ldmz: 33
blts.f:74	16, ranks 0-15	jend: 8 ldmx: 9 j: 9 ldmy: 9 jst: 1-2 ldmz: 33

Figure 40: *Output from Tracepoints in a Fortran application*

Where tracepoints are passed by multiple processes within a short interval, the outputs will be merged. Sparklines of the values recorded are shown for numeric values – along with the range of values obtained – showing the variation across processes.

As alike tracepoints are merged then this can lose the order/causality between *different* processes in tracepoint output. For example, if process 0 passes a tracepoint at time T, and process 1 passes the tracepoint at T + 0.001, then this will be shown as one passing of both process 0 and process 1, with no ordering inferred.

Sequential consistency is preserved during merging, in that for any process, the sequence of tracepoints for that process will be in order.

To find particular values or interesting patterns, use the *Only show lines containing* box at the bottom of the panel. Tracepoint lines matching the text entered here will be shown, the rest will be hidden. To search for a particular value, for example, try “my_var: 34 “ - in this case the space at the end helps distinguish between my_var: 34 and my_var: 345.

For more detailed analysis you may with to export the tracepoints – right-click and choose Export from the pop-up menu. A HTML tracepoint log will be written using the same format as DDT's offline mode.

6.15 Version Control Breakpoints and Tracepoints

Version control breakpoint/tracepoint insertion allows you to quickly record the state of the parts of the target program that were last modified in a particular revision. The resulting tracepoint output may be viewed in the Tracepoint Output tab or the Logbook tab and may be exported or saved as part of a logbook or offline log.

© 2015 Allinea Software Ltd.

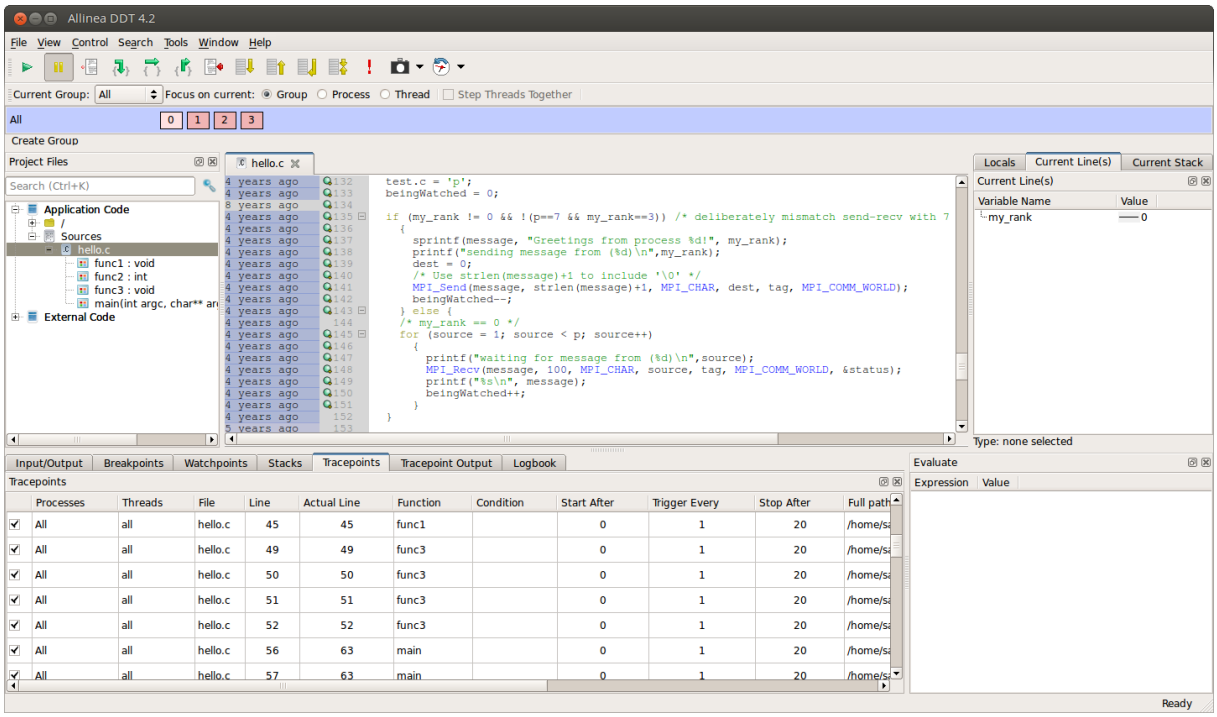


Figure 41: *DDT with version control tracepoints*

Version control tracepoints may be inserted either in the graphical interactive mode or in offline mode via a command line argument.

In interactive mode enable “Version Control Information” from the “View” menu and wait for the annotation column to appear in the code editor (this does not appear for files that are not tracked by a supported version control system).

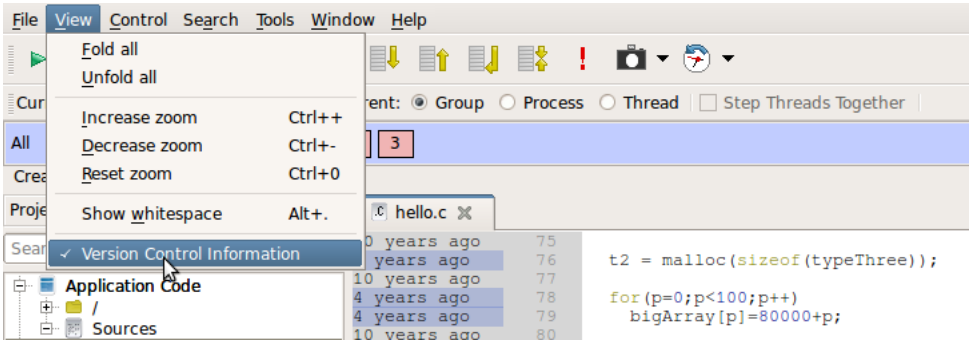


Figure 42: *Version Control - Enable from Menu*

Right click a line last modified by the revision of interest and choose “Trace Variables At This Revision”.

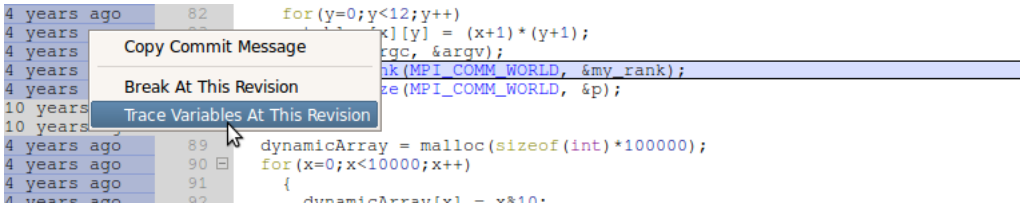


Figure 43: *Version Control - Trace at this*

DDT will find all the source files modified in the revision, detect the variables on the lines modified in the revision and insert tracepoints (pending if necessary). A progress dialog may be shown for lengthy tasks.

Both the tracepoints and the tracepoint output in the Tracepoints, Tracepoint Output, and Logbook tabs may be double-clicked during a session to jump to the corresponding line of source in the code viewer.

In offline mode supply the additional argument `-trace-changes` and DDT will apply the same process as in interactive mode using the current revision of the repository.

By default version control tracepoints are removed after 20 hits. To change this hit limit set the environment variable `DDT_VCS_TRACEPOINT_HIT_LIMIT` to an integer greater than or equal to `0`. To configure version control tracepoints to have no hit limit set this to `0`.

6.16 Examining The Stack Frame

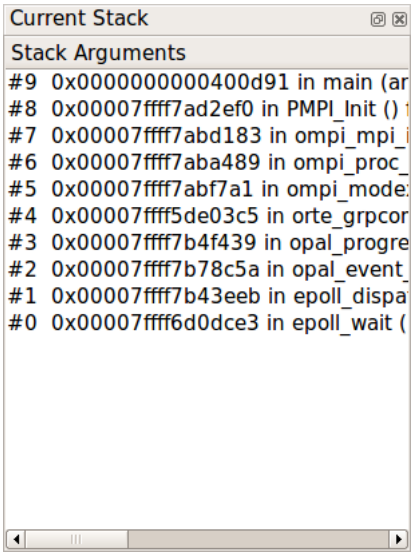


Figure 44: *The Stack Tab*

The stack back trace for the current process and thread are displayed under the *Stack* tab of the *Variables Window*. When you select a stack frame DDT will jump to that position in the code (if it is available) and will display the local variables for that frame. The toolbar can also be used to step up or down the stack, or jump straight to the bottom-most frame.

6.17 Align Stacks

The align stacks button, or CTRL-A hotkey, sets the stack of the current thread on every process in a group to the same level – where possible – as the current process.

This feature is particularly useful where processes are interrupted – by the pause button – and are at different stages of computation. This enables tools such as the *Cross-Process Comparison* window to compare equivalent local variables, and also simplifies casual browsing of values.

© 2015 Allinea Software Ltd.

6.18 ``Where are my processes?'' - Viewing Stacks in Parallel

6.18.1 Overview

To find out where your program is, in one single view, look no further than the *Parallel Stack View*. It's found in the bottom area of DDT's GUI, tabbed alongside *Input/Output*, *Breakpoints* and *Watches*:

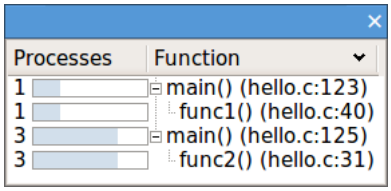


Figure 45: *DDT Parallel Stack View*

Do you want to know where a group's processes are? Click on the group and look at the *Parallel Stack View* – it shows a tree of functions, merged from every process in the group (by default). If there's only one branch in this tree – one list of functions – then all your processes are at the same place. If there are several different branches, then your group has split up and is in different parts of the code! Click on any branch to see its location in the *Source Code Viewer*, or hover your mouse over it and a little popup will list the processes at that location. Right-click on any function in the list and select *New Group* to automatically gather the processes at that function together in a new group, labelled by the function's own name.

The best way to learn about the *Parallel Stack View* is to simply use it to explore your program. Click on it and see what happens. Create groups with it, and watch what happens to it as you step processes through your code. The *Parallel Stack View*'s ability to display and select large numbers of processes based on their location in your code is invaluable when dealing with moderate to large numbers of processes.

6.18.2 The Parallel Stack View in Detail

The *Parallel Stack View* takes over much of the work of the *Stack* display, but instead of just showing the current process, this view combines the call trees (commonly called *stacks*) from many processes and displays them together. The call tree of a process is the list of functions (strictly speaking *frames* or locations within a function) that lead to the current position in the source code. For example, if `main()` calls `read_input()`, and `read_input()` calls `open_file()`, and you stop the program inside `open_file()`, then the call tree will look like this:

```
main()
   read_input()
     open_file()
```

If a function was compiled with debug information (usually `-g`) then DDT adds extra information, telling you the exact source file and line number that your code is on. Any functions without debug information are greyed-out and are not shown by default. Functions without debug information are typically library calls or memory allocation subroutines and are not generally of interest. To see the entire list of functions, right-click on one and choose *Show Children* from the pop-up menu.

You can click on any function to select it as the 'current' function in DDT. If it was compiled with debug information, then DDT will also display its source code in the main window, and its local variables and so on in the other windows.

© 2015 Allinea Software Ltd.

One of the most important features of the *Parallel Stack View* is its ability to show the position of many processes at once. Right-click on the view to toggle between:

1. Viewing all the processes in your program at once
2. Viewing all the processes in the current group at once (default)
3. Viewing only the current process

The function that DDT is currently displaying and using for the variable views is highlighted in dark blue. Clicking on another function in the *Parallel Stack View* will select another frame for the source code and variable views. It will also update the Stack display, since these two controls are complementary. If the processes are at several different locations, then only the current process' location will be shown in dark blue. The other processes' locations will be shown in a light blue:

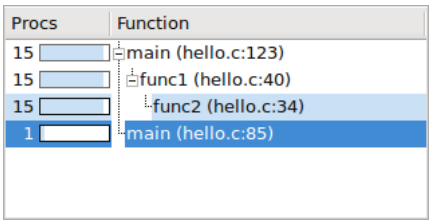


Figure 46: *Current Frame Highlighting in Parallel Stack View*

In the example above, the program's processes are at two different locations. 1 process is in the `main` function, at line 85 of `hello.c`. The other 15 processes are inside a function called `func2`, at line 34 of `hello.c`. The 15 processes reached `func2` in the same way – `main` called func1 on line 123 of `hello.c`, then `func1` called `func2` on line 40 of `hello.c`. Clicking on any of these functions will take you to the appropriate line of source code, and display any local variables in that stack frame.

There are two optional columns in the *Parallel Stack View*. The first, *Procs* shows the number of processes at each location. The second, *Threads*, shows the number of threads at each location. By default, only the number of processes is shown. Right-click to turn these columns on and off. Note that in a normal, single-threaded MPI application, each process has one thread and these two columns will show identical information.

Hovering the mouse over any function in the *Parallel Stack View* displays the full path of the filename, and a list of the process ranks that are at that location in the code:

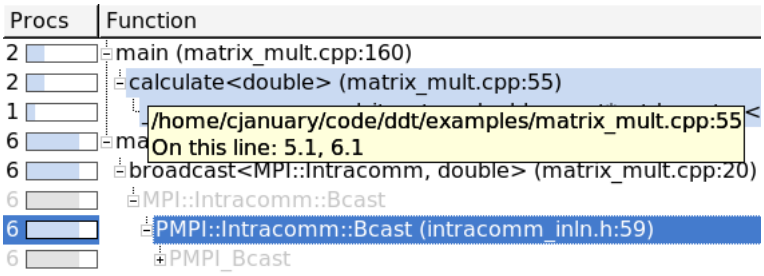


Figure 47: *Parallel Stack View tool tip*

DDT is at its most intuitive when each process group is a collection of processes doing a similar task. The *Parallel Stack View* is invaluable in creating and managing these groups. Simply right-click on any function in the combined call tree and choose the *New Group* option. This will create a new process group that contains only the processes sharing that location in code. By default DDT uses the name of

the function for the group, or the name of the function with the file and line number if it's necessary to distinguish the group further.

6.19 Browsing Source Code

Source code will be automatically displayed – when a process is stopped, when you select a process or change position in the stack. If the source file cannot be found you will be prompted for its location.

DDT highlights lines of the source code to show where your program currently is. Lines that contain processes from the current group are shaded in that group's colour. Lines only containing processes from other groups are shaded in grey.

This pattern is repeated in the focus on process and thread modes. For example, when you focus on a process, DDT highlights lines containing that process in the group colour, and other processes from that group in grey.

DDT also highlights lines of code that are on the stack – functions that your program will return to when it has finished executing the current one. These are drawn with a faded look to distinguish them from the currently-executing lines.

You can hover the mouse over any highlighted line to see which processes/threads are currently on that line. This information is presented in a variety of ways, depending on the current focus setting:

Focus on Group

A list of groups that are on the selected line, along with the processes in them on this line, and a list of threads from the current process on the selected line.

Focus on Process

A list of the processes from the current group that are on this line, along with the threads from the current process on the selected line.

Focus on Thread

A list of threads from the current process on the selected line.

The tool tip distinguishes between processes and threads that are currently executing that line, and ones that are on the stack by grouping them under the headings *On the stack* and *On this line*.

Variables and Functions

Right-clicking on a variable or function name in the *Source Code Viewer* will make DDT check whether there is a matching variable or function, and then display extra information and options in a sub-menu.

In the case of a variable, the type and value are displayed, along with options to view the variable in the *Cross-Process Comparison Window* (CPC) or the *Multi-Dimensional Array Viewer* (MDA), or to drop the variable into the *Evaluate Window* – each of which are described in the next chapter.

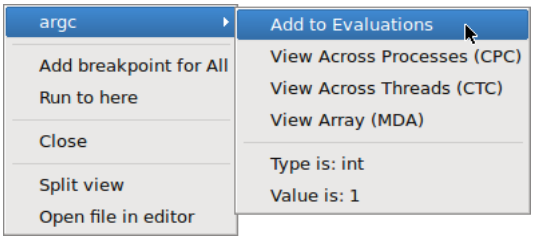


Figure 48: *Right-Click Menu – Variable Options*

© 2015 Allinea Software Ltd.

In the case of a function, it is also possible to add a breakpoint in the function, or to the source code of the function when available.

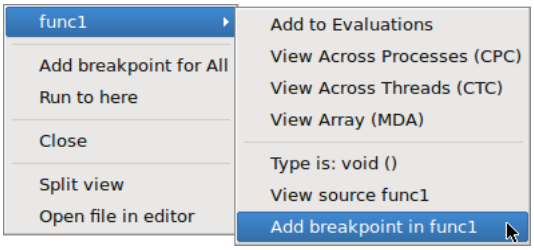


Figure 49: *Right-Click Menu – Function Options*

6.20 Simultaneously Viewing Multiple Files

DDT presents a tabbed pane view of source files, but occasionally it may be useful to view two files simultaneously – whilst tracking two different processes for example.

Inside the code viewing panel, right-click to split the view. This will bring a second tabbed pane which can be viewed beneath the first one. When viewing further files, the currently 'active' panel will display the file. Click on one of the views to make it active.

The split view can be reset to a single view by right-clicking in the code panel and deselecting the split view option.

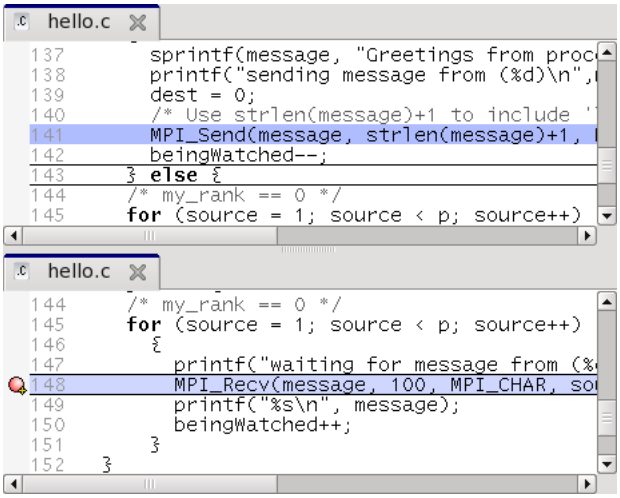

hello.c
137 sprintf(message, "Greetings from proc
138 printf("sending message from (%d)\n",
139 dest = 0;
140 /* Use strlen(message)+1 to include '
141 MPI_Send(message, strlen(message)+1,
142 beingWatched--;
143 } else {
144 /* my_rank == 0 */
145 for (source = 1; source < p; source++)
hello.c
144 /* my_rank == 0 */
145 for (source = 1; source < p; source++)
146 {
147 printf("waiting for message from (%
148 MPI_Recv(message, 100, MPI_CHAR, so
149 printf("%s\n", message);
150 beingWatched++;
151 }
152 }

Figure 50: *Horizontal Alignment Of Multiple Source Files*

6.21 Signal Handling

By default DDT will stop a process if it encounters one of the standard signals (but see section 6.21.1 Custom Signal Handling (Signal Dispositions) below). For example:

- `SIGSEGV` – Segmentation fault

  The process has attempted to access memory that is not valid for that process. Often this will be caused by reading beyond the bounds of an array, or from a pointer that has not been allocated yet. The DDT *Memory Debugging* feature may help to resolve this problem.

© 2015 Allinea Software Ltd.

- `SIGFPE` – Floating Point Exception

  This is raised typically for integer division by zero, or dividing the most negative number by -1. Whether or not this occurs is Operating System dependent, and not part of the POSIX standard. Linux platforms will raise this.

  Note that floating point division by zero will not necessarily cause this exception to be raised, behaviour is compiler dependent. The special value `Inf` or `-Inf` may be generated for the data, and the process would not be stopped.

- `SIGPIPE` – Broken Pipe

  A broken pipe has been detected whilst writing.

- `SIGILL` – Illegal Instruction

`SIGUSR1`, `SIGUSR2`, `SIGCHLD`, `SIG63` and `SIG64` are passed directly through to the user process without being intercepted by DDT.

6.21.1 Custom Signal Handling (Signal Dispositions)

You can change the way individual signals are handled using the *Signal Handling* window. To open the window select the *Control* → *Signal* Handling... menu item.

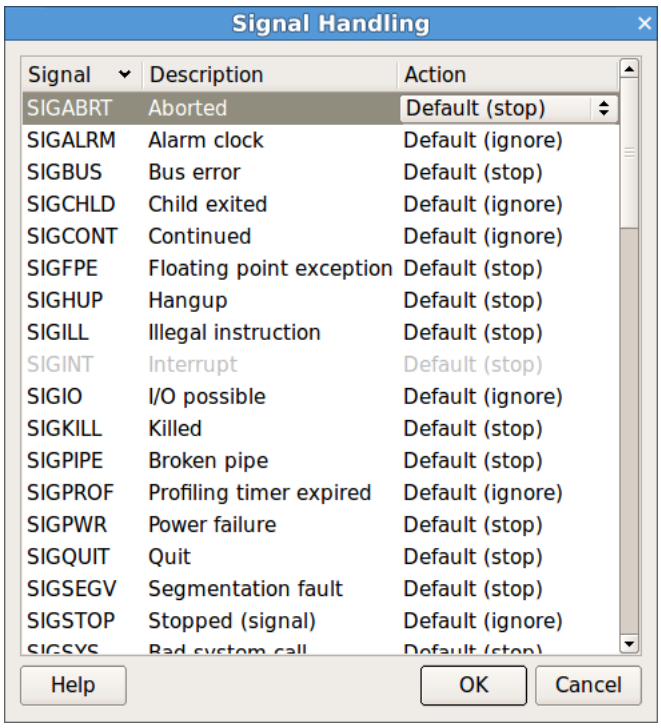


Figure 51: *Signal Handling dialog*

Set a signal's action to *Stop* to stop a process whenever it encounters the given signal, or *Ignore* to let the process receive the signal and continue playing without being stopped by the debugger.

6.21.2 Sending Signals

The *Send Signal* window (select the *Control* → *Send Signal*... menu item) allows a signal to be sent to the debugged processes. Select the signal you want to send from the drop-down list and click the Send

to process button.

7 DDT: Viewing Variables And Data

The *Variables Window* contains two tabs that provide different ways to list your variables. The *Locals* tab contains all the variables for the current stack frame, while the *Current Line(s)* tab displays all the variables referenced on the currently selected lines.

Right clicking in these windows brings up additional options – including the ability to edit values (in the Evaluations window), to change the display base, or to compare data across processes and threads. The right-click menu will also allow you to choose whether the fields in structures (classes or derived types) should be displayed alphabetically by element name or not – which is useful for when structures have very many different fields.

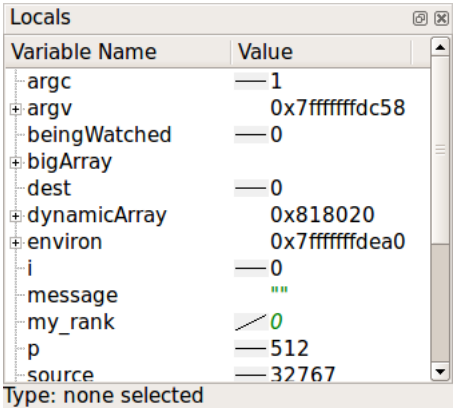


Figure 52: *Displaying Variables*

7.1 Sparklines

Numerical values may have sparklines displayed next to them. A sparkline is a line graph of process rank against value of the related expression. The graph is bound by the minimum and maximum values found, or in the case that all values are equal the line is drawn across the vertical center of the highlighted region. Erroneous values such as *Nan* and *Inf* are represented as red, vertical bars. Clicking on a sparkline will display the Cross-Process Comparison window for closer analysis.

7.2 Current Line

You can select a single line by clicking on it in the code viewer - or multiple lines by clicking and dragging. The variables are displayed in a tree view so that user-defined classes or structures can be expanded to view the variables contained within them. You can drag a variable from this window into the *Evaluate Window*; it will then be evaluated in whichever stack frame, thread or process you select.

7.3 Local Variables

The *Locals* tab contains local variables for the current process's currently active thread and stack frame.

For Fortran codes the amount of data reported as local can be substantial – as this can include many global or common block arrays. Should this prove problematic, it is best to conceal this tab underneath the *Current Line(s)* tab as this will not then update after ever step.

© 2015 Allinea Software Ltd.

It is worth noting that variables defined within common blocks may not appear in the local variables tab with some compilers, this is because they are considered to be global variables when defined in a common memory space.

The *Locals* view compares the value of scalar variables against other processes. If a value varies across processes in the current group the value is highlighted in green.

When stepping or switching processes if the value of a variable is different from the previous position or process it is highlighted in blue.

After stepping out of function the return value is displayed at the top of the *Locals* view (for selected debuggers).

7.4 Arbitrary Expressions And Global Variables

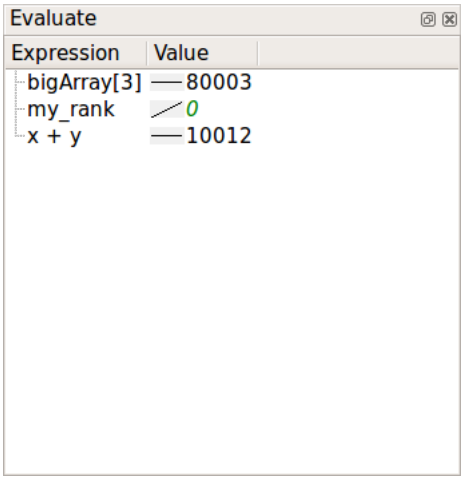


Figure 53: *Evaluating Expressions*

Since the global variables and arbitrary expressions do not get displayed with the local variables, you may wish to use the *Current Line(s)* tab in the *Variables* window and click on the line in the *Source Code Viewer* containing a reference to the global variable.

Alternatively, the *Evaluate* panel can be used to view the value of any arbitrary expression. Right-click on the *Evaluate* window, click on *Add Expression*, and type in the expression required in the current source file language. This value of the expression will be displayed for the current process and stack/thread, and is updated after every step.

Note: at the time of writing DDT does not apply the usual rules of precedence to logical Fortran expressions, such as `x .ge. 32 .and. x .le. 45`. For now, please bracket such expressions thoroughly: `(x .ge. 32) .and. (x .le. 45)`. It is also worth noting that although the Fortran syntax allows you to use keywords as variable names, DDT will not be able to evaluate such variables on most platforms. Please contact support@allinea.com if this is a problem for you.

© 2015 Allinea Software Ltd.

7.4.1 Fortran Intrinsics

The following Fortran intrinsics are supported by the default GNU debugger included with DDT:

ABS	AIMAG	CEILING	CMPLX
FLOOR	IEEE_IS_FINITE	IEEE_IS_INF	IEEE_IS_NAN
IEEE_IS_NORMAL	ISFINITE	ISINF	ISNAN
ISNORMAL	MOD	MODULO	REALPART

Support in other debuggers, including the CUDA debugger variants, may vary.

7.4.2 Changing the language of an Expression

Ordinarily, expressions in the Evaluate window and Locals/Current windows are evaluated in the language of the current stack frame. This may not always be appropriate – for example a pointer to user defined structure may be passed as value within a Fortran section of code, and you may wish to view the fields of the C structure. Alternatively, you may wish to view a global value in a C++ class whilst your process is in a Fortran subroutine.

You can change the language that DDT uses for your expressions by right clicking on the expression, and clicking *Change Type/Language* - selecting the appropriate language for the expression. To restore the default behaviour, change this back to *Auto*.

7.4.3 Macros and #defined Constants

By default, many compilers will not output sufficient information to allow the debugger to display the values of "#defined" constants or macros – as including this information can greatly increase executable sizes.

With the GNU compiler, adding the "`-g3`" option to the command line options will generate extra definition information which DDT will then be able to display.

7.5 Help With Fortran Modules

An executable containing Fortran modules presents a special set of problems for developers:

- If there are many modules, each of which contains many procedures and variables (each of which can have the same name as something else in a separate Fortran module), keeping track of which name refers to which entity can become difficult
- When the *Locals* or *Current Line(s)* tabs (within the *Variables* window) display one of these variables, to which Fortran module does the variable belong?
- How do you refer to a particular module variable in the *Evaluate* window?
- How do you quickly jump to the source code for a particular Fortran module procedure?

To help with this, DDT provides a *Fortran Modules* tab in the *Project Navigator* window.

When DDT begins a session, Fortran module membership is automatically found from the information compiled into the executable.

A list of Fortran modules found is displayed in a simple tree view within the *Fortran Modules* tab of the *Project Navigator* window.

Each of these modules can be 'expanded' (by clicking on the + symbol to the left of the module name) to display the list of member procedures, member variables and the current values of those member variables.

Clicking on one of the displayed procedure names will cause the *Source Code Viewer* to jump to that procedure's location in the source code. In addition, the return-type of the procedure will be displayed at the bottom of the *Fortran Modules* tab – Fortran subroutines will have a return-type of `VOID ()`.

Similarly, clicking on one of the displayed variable names will cause the type of that variable to be displayed at the bottom of the *Fortran Modules* tab.

A module variable can be dragged and dropped into the *Evaluate* window. Here, all of the usual *Evaluate* window functionality applies to the module variable. To help with variable identification in the *Evaluate* window, module variable names are prefixed with the Fortran module name and two colons ::.

Right-clicking within the *Fortran Modules* tab will bring up a context menu. For variables, choices on this menu will include sending the variable to the *Evaluate* window, the *Multi-Dimensional Array Viewer* and the *Cross-Process Comparison Viewer*.

Some caveats apply to the information displayed within the *Fortran Modules* tab:

1. The *Fortran Modules* tab will not be displayed if the underlying debugger does not support the retrieval and manipulation of Fortran module data.
2. The *Fortran Modules* tab will display an empty module list if the Fortran modules debug data is not present or in a format understood by DDT.

One limitation of the *Fortran Modules* tab is that the modules debug data compiled into the executable does not include any indication of the module `USE` hierarchy (e.g. if module A `USE`s module B, the inherited members of module B are not shown under the data displayed for module A). Consequently, the *Fortran Modules* tab shows the module `USE` hierarchy in a flattened form, one level deep.

7.6 Viewing Complex Numbers in Fortran

When working with complex numbers, you may wish to view only the real or imaginary elements of the number. This can useful when evaluating expressions, or viewing an array in the Multi Dimensional Array Viewer (See section 7.15 Multi Dimensional Array Viewer (MDA)).

You can use the Fortran intrinsic functions `REALPART` and `AIMAG` to get the real or imaginary parts of a number, or their C99 counterparts `creal` and `cimag`.

Complex numbers in Fortran can also be accessed as an array, where element 1 is the real part, and element 2 is the imaginary part.

© 2015 Allinea Software Ltd.

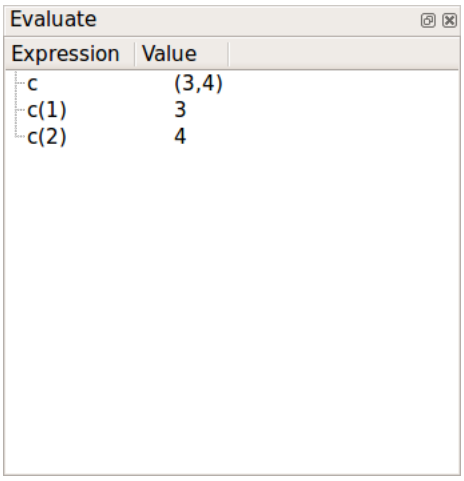


Figure 54: *Viewing the Fortran complex number 3+4i*

7.7 C++ STL Support

DDT uses pretty printers for the GNU C++ STL implementation and Nokia's Qt library, and Boost, designed for use with the GNU Debugger. These are used automatically to present such C++ data in a more understandable format.

For some compilers, the STL pretty printing can be confused by non-standard implementations of STL types used by a compiler's own STL implementation. In this case, and in the case where you wish to see the underlying implementation of an STL type, you can disable pretty printing by running DDT with the environment variable setting DDT_DISABLE_PRETTY_PRINT=1.

7.8 Custom Pretty Printers

In addition to the pre-installed pretty printers you may also use your own GDB pretty printers.

Note: custom pretty printers are only supported when using the GDB 7.6.2 debugger. You must select this debugger on the System Settings page of the Options window.

A GDB pretty printer consists of an `auto-load` script that is automatically loaded when a particular executable or shared object is loaded and the actual pretty printer Python classes themselves. To make a pretty printer available in DDT copy it to `~/.allinea/gdb`.

7.8.1 Example

An example pretty printer may be found in {installation-directory}/examples.

Compile the `fruit` example program using the GNU C++ compiler as follows:

```
cd {installation-directory}/examples
make -f fruit.makefile
```

Now start DDT with the example program as follows:

```
ddt -start {installation-directory}/examples/fruit
```

After the program has started right-click on line 20 and click the *Run to here* menu item. Click on the *Locals* tab and notice that the internal variable of `myFruit` are displayed.

© 2015 Allinea Software Ltd.

Now install the fruit pretty printer by copying the files to `~/.allinea/gdb` as follows:

```
cp -r {installation-directory}/examples/fruit-pretty-printer/* ~/.
   allinea/gdb/
```

Re-run the program in DDT and run to line 20, as before. Click on the *Locals* tab and notice that now, instead of the internal variable of `myFruit`, the type of fruit is displayed instead.

7.9 Viewing Array Data

Fortran users may find that it is not possible to view the upper bounds of an array. This is due to a lack of information from the compiler. In these circumstances DDT will display the array with a size of 0, or simply `<unknown_bounds>`. It is still possible to view the contents of the array however using the Evaluate window to view `array(1)`, `array(2)`, etc. as separate entries.

To tell DDT the size of the array right-click on the array and select the *Edit Type...* menu option. This will open a window like the one below. Enter the real type of the array in the *New Type* box.

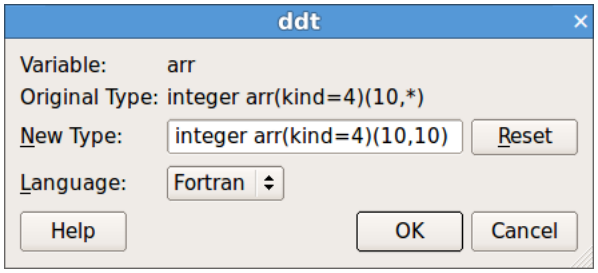


Figure 55: *Edit Type window*

Alternatively the MDA can be used to view the entire array.

7.10 UPC Support

Allinea DDT supports many different UPC compilers – including the GNU UPC compiler, the Berkeley UPC compiler and those provided by Cray and SGI. Note that in order to enable UPC support, you may need to select the appropriate MPI/UPC implementation from DDT's Options/System menu. See Section 4.1.5 UPC

Debugging UPC applications introduces a small number of changes to the user interface.

- Processes will be identified as UPC Threads, this is purely a terminology change for consistency with the UPC language terminology. UPC Threads will have behaviour identical to that of separate processes: groups, process control and cross-process data comparison for example will apply across UPC Threads.
- The type qualifier `shared` is given for shared arrays or pointers to shared.
- Shared pointers are printed as a triple (address, thread, phase). For indefinitely blocked pointers the phase is omitted.
- Referencing shared items will yield a shared pointer and pointer arithmetic may be performed on shared pointers.
- Dereferencing a shared pointer (e.g. dereferencing `*(&x[n] + 1])`) will correctly evaluate and fetch remote data where required.

© 2015 Allinea Software Ltd.

- Values in shared arrays are not automatically compared across processes: the value of `x[i]` is by definition identical across all processes. It is not possible to identify pending read/write to remote data. Non-shared data types such as local data or local array elements will still be compared automatically.
- Distributed arrays are handled implicitly by the debugger. There is no need to use the explicit *distributed dimensions* feature in the MDA.

All other components of DDT will be identical to debugging any multi-process code.

7.11 Changing Data Values

In the *Evaluate* window, the value of an expression may be set by right-clicking and selecting *Edit Value*. This will allow you to change the value of the expression for the current process, current group, or for all processes.

Note: The variable must exist in the current stack frame for each process you wish to assign the value to.

7.12 Viewing Numbers In Different Bases

When you are viewing an integer numerical expression you may right-click on the value and use the *View As* sub menu to change which base the value is displayed in. The *View As* → *Default* option displays the value in its original (default) base.

7.13 Examining Pointers

You can examine pointer contents by clicking the + next to the variable or expression. This will automatically dereference the pointer. You can also use the *View As Vector*, *Reference*, or *Dereference* menu items.

7.14 Multi-Dimensional Arrays in the Variable View

When viewing a multi-dimensional array in either the *Locals*, *Current Line(s)* or *Evaluate* windows it is possible to expand the array to view the contents of each cell. In C/C++ the array will expand from left to right (`x`, `y`, `z` will be seen with the x column first, then under each `x` cell a `y` column etc.) whereas in Fortran the opposite will be seen with arrays being displayed from right to left as you read it (so `x`, `y`, `z` would have `z` as the first column with `y` under each `z` cell etc.)

© 2015 Allinea Software Ltd.

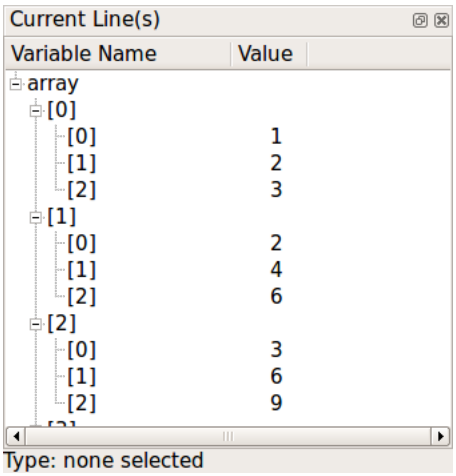


Figure 56: *2D Array In C: type of tables int[12][12]*

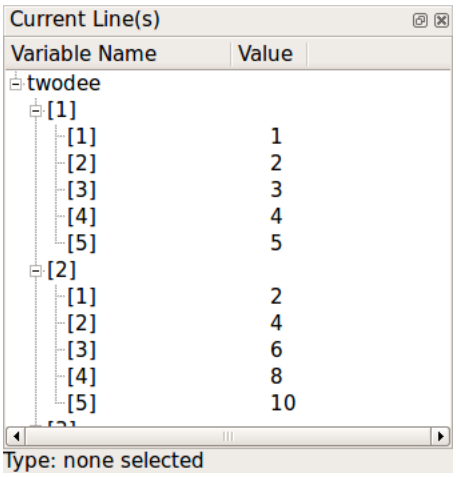


Figure 57: *2D Array In Fortran: type of twodee is integer(3,5)*

7.15 Multi Dimensional Array Viewer (MDA)

DDT provides a *Multi-Dimensional Array (MDA) Viewer* (fig. 58) for viewing multi-dimensional arrays.

To open the *Multi-Dimensional Array Viewer*, right-click on a variable in the *Source Code*, *Locals*, *Current Line(s)* or *Evaluate* views and select the *View Array (MDA)* context menu option. You can also open the MDA directly by selecting the *Multi-Dimensional Array Viewer* menu item from the *View* menu.

© 2015 Allinea Software Ltd.

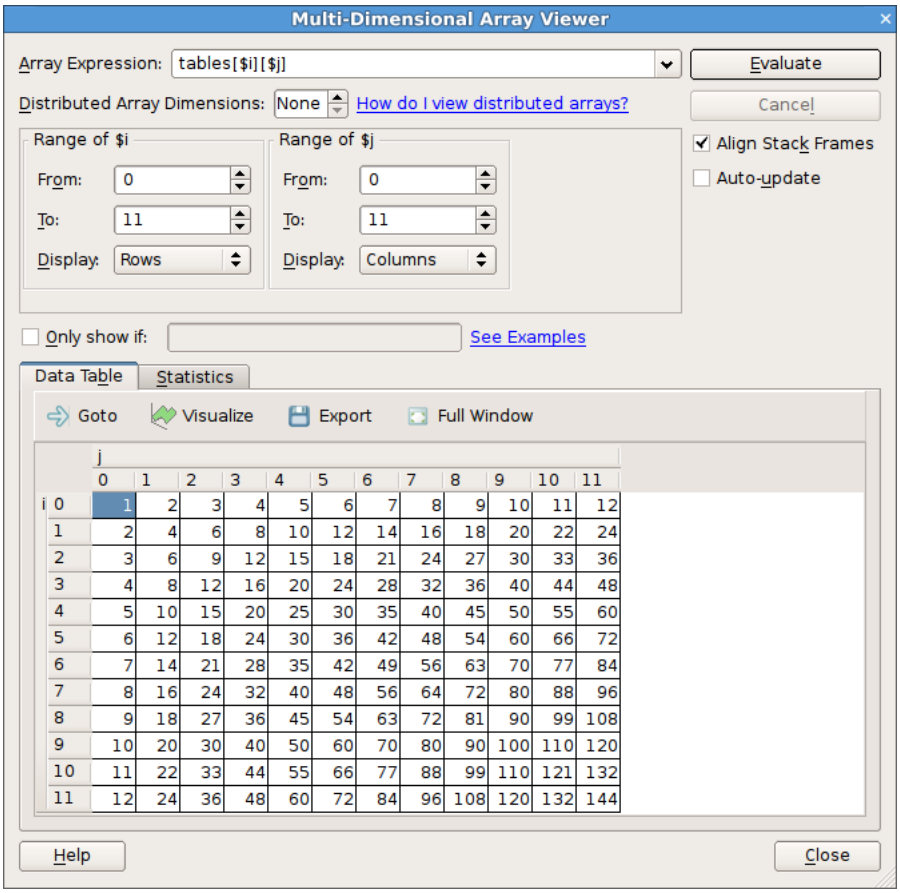


Figure 58: *Multi-Dimensional Array Viewer*

If you open the MDA by right clicking on a variable, DDT will automatically set the *Array Expression* and other parameters based on the type of the variable. Click the *Evaluate* button to see the contents of the array in the *Data Table*.

The *Full Window* button hides the settings at the top of the window so the table of values occupies the full window, allowing you to make full use of your screen space. Click the button again to reveal the settings again.

7.15.1 Array Expression

The *Array Expression* is an expression containing a number of *subscript metavariables* that are substituted with the subscripts of the array. For example, the expression `myArray($i, $j)` has two metavariables, `$i` and `$j`. The metavariables are unrelated to the variables in your program.

The range of each metavariable is defined in the boxes below the expression, e.g. *Range of* $*i*. The *Array Expression* is evaluated for each combination of `$i`, `$j`, etc. and the results shown in the *Data Table*. You can also control whether each metavariable is shown in the *Data Table* using Rows or Columns.

The metavariables may be re-ordered by dragging and dropping them. For C/C++ expressions the major dimension is on the left and the minor dimension on the right, for Fortran expressions the major dimension is on the right and the minor dimension on the left. Distributed dimensions may not be re-ordered – they must always be the most major dimensions.

© 2015 Allinea Software Ltd.

7.15.2 Filtering by Value

You may want the *Data Table* to only show elements that fit a certain criteria, e.g. elements that are zero. If the *Only show* if box is checked then only elements that match the boolean expression in the box are displayed in the *Data Table*, e.g. `$value == 0`. The special metavariable `$value` in the expression is replaced by the actual value of each element. The *Data Table* automatically hides rows or columns in the table where no elements match the expression.

7.15.3 Distributed Arrays

A distributed array is an array that is distributed across one or more processes as local arrays.

The Multi-Dimensional Array Viewer can display certain types of distributed arrays, namely UPC shared arrays (for supported UPC implementations), and general arrays where the distributed dimensions are the most major (i.e. the distributed dimensions change the most slowly) and are independent from the non-distributed dimensions.

UPC shared arrays are treated the same as local arrays, simply right-click on the array variable and select View Array (MDA).

To view a non-UPC distributed array first create a process group containing all the processes that the array is distributed over. If the array is distributed over all processes in your job then you can simply select the *All* group instead. Right-click on the local array variable in the *Source Code*, *Locals*, *Current Line(s)* or *Evaluate* views. The *Multi-Dimensional Array Viewer* window will open with the Array Expression already filled in. Enter the number of distributed array dimensions in the corresponding box. A new subscript metavariable (`$p`, `$q`, etc.) will be automatically added for each distributed dimension. Enter the ranges of the distributed dimensions so that the product is equal to the number of processes in the current process group, then click the *Evaluate* button.

7.15.4 Advanced: How Arrays Are Laid Out in the Data Table

The Data Table is two dimensional, but the Multi-Dimensional Array Viewer may be used to view arrays with any number of dimensions, as the name implies. This section describes how multi-dimensional arrays are displayed in the two dimensional table.

Each subscript metavariable (`$i`, `$j`, `$p`, `$q`, etc.) maps to a separate dimension on a hypercube. Usually the number of metavariables is equal to the number of dimensions in a given array, but this does not necessarily need to be the case, e.g. `myArray($i, $j) * $k` introduces an extra dimension, `$k`, as well as the two dimensions corresponding to the two dimensions of `myArray`.

The figure below corresponds to the expression `myArray($i, $j)` with `$i = 0..3` and `$j = 0..4`.

© 2015 Allinea Software Ltd.

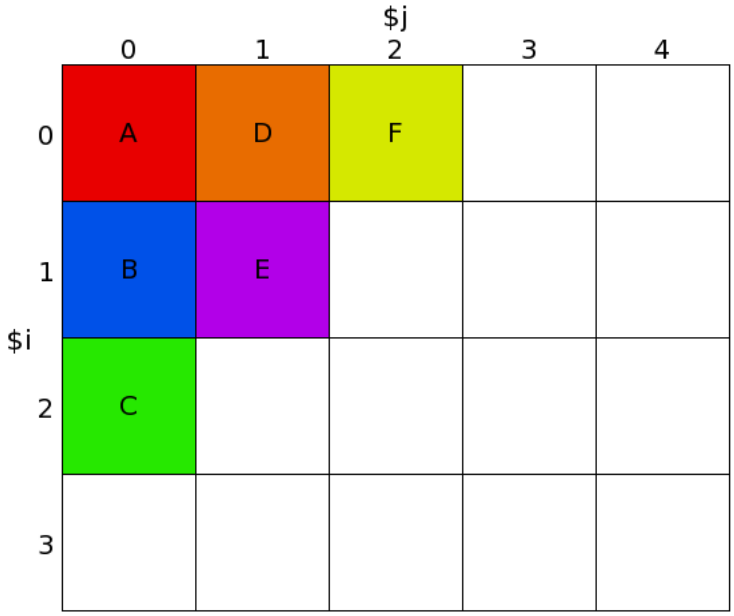


Figure 59: *myArray($i, $j) with $i = 0..3 and $j = 0..4.*

Now let's say `myArray` is part of a three dimensional array distributed across three processes. The figure below shows what the local arrays look like for each process.

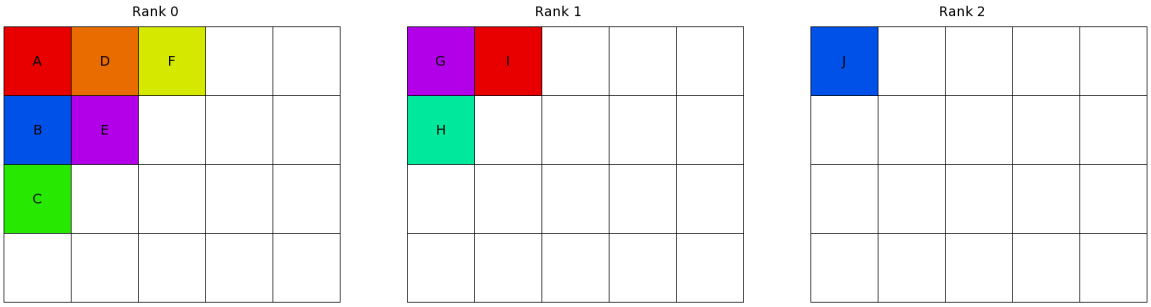


Figure 60: *The local array myArray($i, $j) with $i = 0..3 and $j = 0..4 on ranks 0 – 2*

And as a three dimensional distributed array with `$p` the distributed dimension:

© 2015 Allinea Software Ltd.

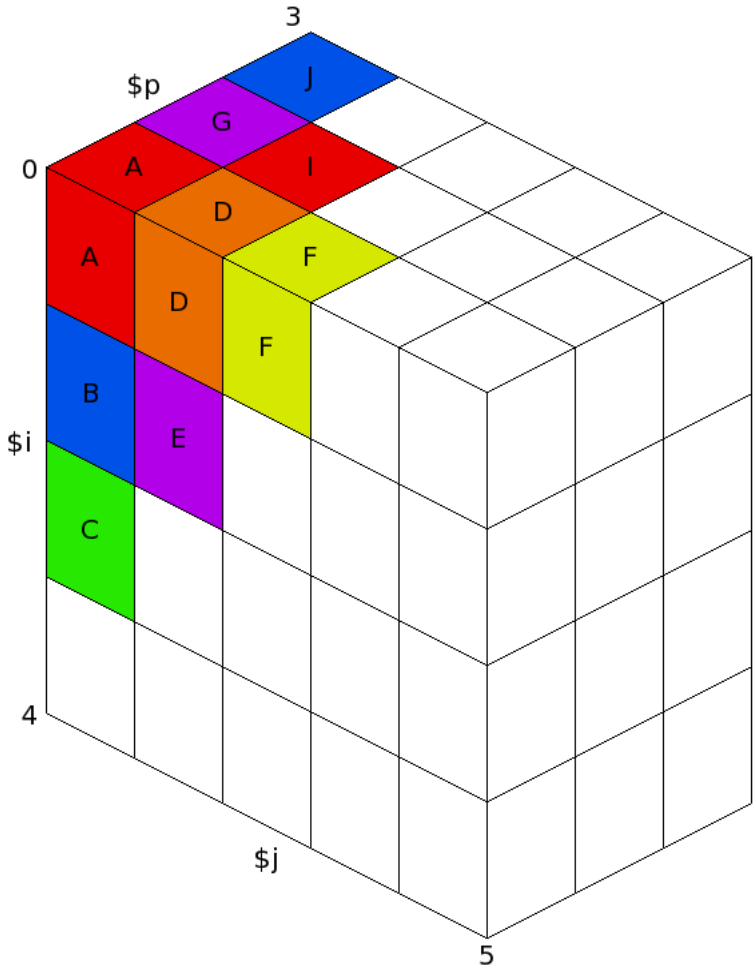


Figure 61: *A three dimensional distributed array comprised of the local array myArray($i, $j) with $i = 0..3 and $j = 0..4 on ranks 0 – 2 with $p the distributed dimension*

This cube is projected (just like 3D projection) onto the two dimensional Data Table. Dimensions marked *Display as Rows* are shown in rows, and dimensions marked Display as Columns are shown in columns, as you would expect.

More than one dimension may viewed as Rows, or more than one dimension viewed as Columns. The dimension that changes fastest depends on the language your program is written in. For C/C++ programs the leftmost metavariable (usually `$i` for local arrays or `$p` for distributed arrays) changes the most slowly (just like with C array subscripts). The rightmost dimension changes the most quickly. For Fortran programs the order is reversed (the rightmost is most major, the leftmost most minor).

The figure below shows how the three dimensional distributed array above is projected onto the two dimensional Data Table:

© 2015 Allinea Software Ltd.

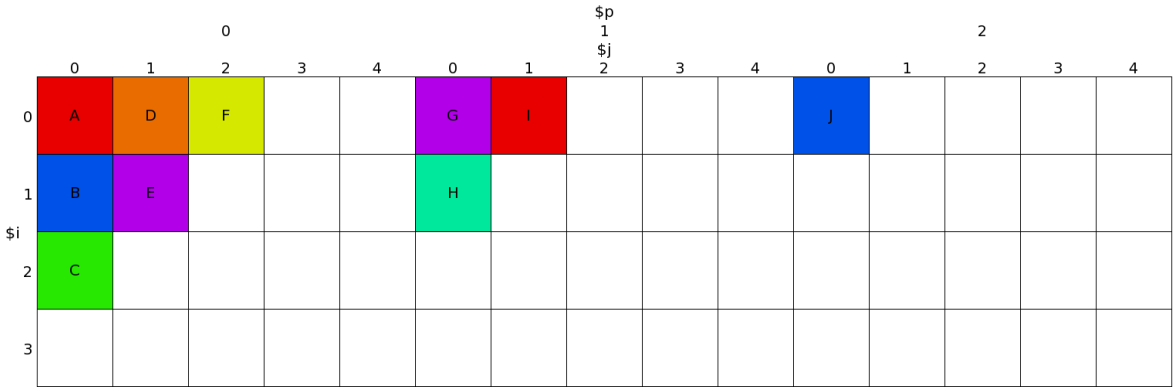


Figure 62: *A three dimensional distributed array comprised of the local array myArray($i, $j) with $i = 0..3 and $j = 0..4 on ranks 0 – 2 projected onto the Data Table with $p (the distributed dimension) and $j displayed as Columns and $i displayed as Rows.*

7.15.5 Auto Update

If you check the *Auto Update* check box the *Data Table* will be automatically updated as you switch between processes/threads and step through the code.

7.15.6 Statistics

The *Statistics* tab displays information which may be of interest, such as the range of the values in the table, and the number of special numerical values, such as `nan` or `inf`.

7.15.7 Export

You may export the contents of the results table to a file in the Comma Separated Values (CSV) or HDF5 format that can be plotted or analysed in your favourite spreadsheet or mathematics program.

There are two CSV export options: List (one row per value) and Table (same layout as the on screen table).

Note: If you export a Fortran array from DDT in HDF5 format the contents of the array are written in column major order. This is the order expected by most Fortran code, but the arrays will be transposed if read with the default settings by C-based HDF5 tools. Most HDF5 tools have an option to switch between row major and column major order.

7.15.8 Visualization

If your system is OpenGL-capable then a 2-D slice of an array, or table of expressions, may be displayed as a surface in 3-D space through the *Multi-Dimensional Array (MDA) Viewer*. You can only plot one or two dimensions at a time – if your table has more than two dimensions the *Visualise* button will be disabled. After filling the table of the *MDA Viewer* with values (see previous section), click *Visualise* to open a 3-D view of the surface. To display surfaces from two or more different processes on the same plot simply select another process in the main process group window and click *Evaluate* in the MDA window, and when the values are ready, click *Visualize* again. The surfaces displayed on the graph may be hidden and shown using the check boxes on the right-hand side of the window.

© 2015 Allinea Software Ltd.

The graph may be moved and rotated using the mouse and a number of extra options are available from the window toolbar.

The mouse controls are:

- Hold down the left button and drag the mouse to rotate the graph.
- Hold down the right button to zoom – drag the mouse forwards to zoom in and backwards to zoom out.
- Hold the middle button and drag the mouse to move the graph.

Please note: DDT requires OpenGL to run. If your machine does not have hardware OpenGL support, software emulation libraries such as MesaGL are also supported.

In some configurations OpenGL is known to crash – a work around if the 3D visualization crashes is to set the environment variable LIBGL_ALWAYS_INDIRECT to 1 – the precise configuration which triggers this problem is not known.

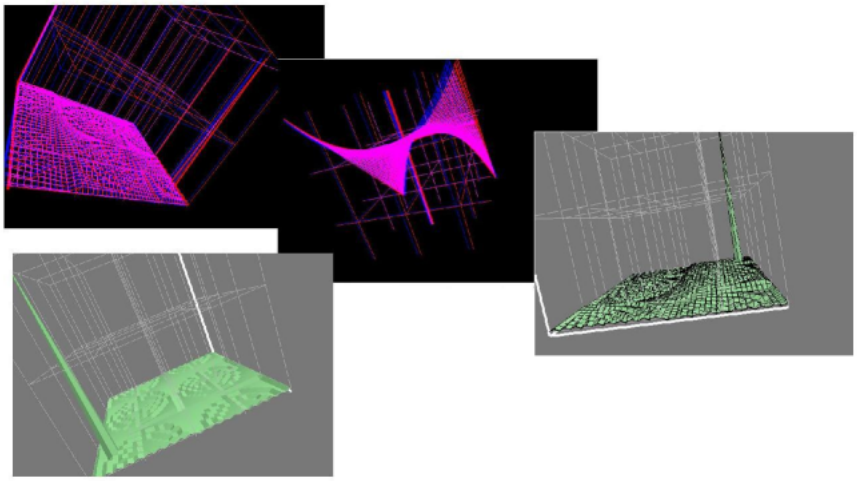


Figure 63: *DDT Visualization*

The toolbar and menu offer options to configure lighting and other effects, including the ability to save an image of the surface as it currently appears. There is even a stereo vision mode that works with red-blue glasses to give a convincing impression of depth and form. Contact Allinea if you need to get hold of some 3D glasses.

7.16 Cross-Process and Cross-Thread Comparison

The *Cross-Process Comparison* and *Cross-Thread Comparison* windows can be used to analyse expressions calculated on each of the processes in the current process group. Each window displays information in three ways: raw comparison, statistically, and graphically.

This is a more detailed view than the sparklines that are automatically drawn against a variable in the evaluations and locals/current line windows for multi-process sessions.

To compare values across processes or threads, right-click on a variable inside the *Source Code*, *Locals*, *Current Line(s)* or *Evaluate* windows and then choose one of the *View Across Processes (CPC)* or *View Across Threads (CTC)* options. You can also bring up the CPC or CTC directly from the *View* menu in the main menu bar. Alternatively, clicking on a sparkline will bring up the CPC.

© 2015 Allinea Software Ltd.

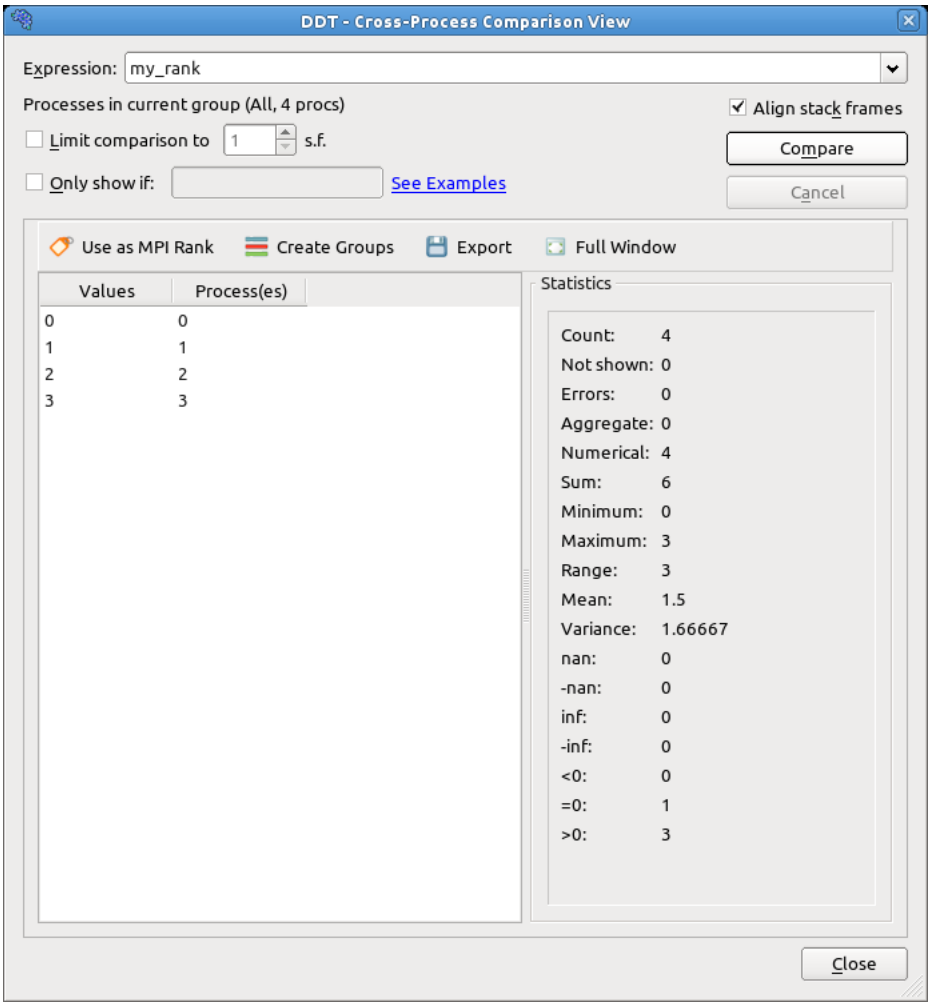


Figure 64: *Cross-Process Comparison – Compare View*

Processes and threads are grouped by expression value when using the raw comparison. The precision of this grouping can be specified (for floating point values) by filling the *Limit* box.

If you are comparing across processes, you can turn each of these groupings of processes into a DDT process group by clicking the create groups button. This will create several process groups – one for each line in the panel. Using this capability large process groups can be managed with simple expressions to create groups. These expressions are any valid expression in the present language (i.e. C/C++/Fortran).

For threaded applications, when using the CTC, if Allinea DDT is able to identify OpenMP thread IDs, a third column will also display the corresponding OpenMP thread IDs for each thread that has each value.

You can enter a second boolean expression in the *Only show* if box to control which values are displayed. Only values for which the boolean expression evaluates to `true` / `.TRUE.` are displayed in the results table. The special metavariable `$value` in the expression is replaced by the actual value. Click the *Show Examples* link to see examples.

The *Align Stack Frames* check box tries to automatically make sure all processes and threads are in the same stack frame when comparing the variable value. This is very helpful for most programs, but you may wish to disable it if different processes/threads run entirely different programs.

The *Use as MPI Rank* button is described in the next section, *Assigning MPI Ranks*.

You can create a group for the ranks corresponding to each unique value by clicking the *Create Groups*

button.

The *Export* button allows you to export the list of values and corresponding ranks as a Comma Separated Values (CSV) file.

The *Full Window* button hides the settings at the top of the window so the list of values occupies the full window, allowing you to make full use of your screen space. Click the button again to rev e al the settings again.

The *Statistics* panel shows Maximum, Minimum, Variance and other statistics for numerical values.

7.17 Assigning MPI Ranks

Sometimes, DDT cannot detect the MPI rank for each of your processes. This might be because you are using an experimental MPI version, or because you have attached to a running program, or only part of a running program. Whatever the reason, it is easy to tell DDT what each process should be called.

To begin, choose a variable that holds the MPI world rank for each process, or an expression that calculates it. Use the *Cross-Process Comparison* window to evaluate the expression across **all** the processes. If the variable is valid, the *Use as MPI Rank* button will be enabled. Click on it; DDT will immediately relabel all its processes with these new values.

What makes a variable or expression valid? These criteria must be met:

1. It must be an integer
2. Every process must have a unique number afterwards

These are the only restrictions. As you can see, there is no need to use the MPI rank if you have an alternate numbering scheme that makes more sense in your application. In fact you can relabel only a few of the processes and not all, if you prefer, so long as afterwards **every** process still has a unique number.

7.18 Viewing Registers

To view the values of machine registers on the currently selected process, select the *Registers* window from the *View* pull-down menu. These values will be updated after each instruction, change in thread or change in stack frame.

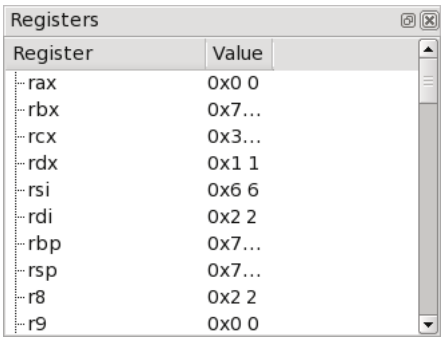


Figure 65: *Register View*

7.19 Interacting Directly With The Debugger

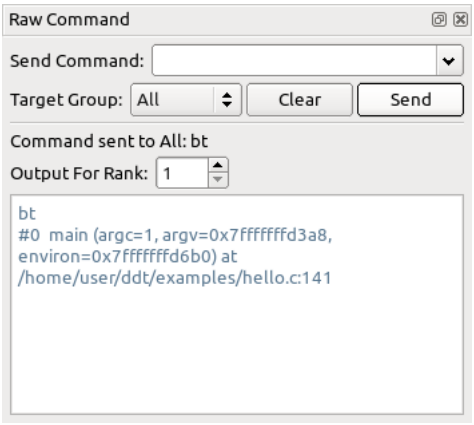


Figure 66: *Raw Command Window*

DDT provides a *Raw Command* window that will allow you to send commands directly to the debugger interface. This window bypasses DDT and its book-keeping - if you set a breakpoint here, DDT will not list this in the breakpoint list, for example.

Be careful with this window; we recommend you only use it where the graphical interface does not provide the information or control you require. Sending commands such as `quit` or `kill` may cause the interface to stop responding to DDT.

Each command is sent to a group of processes (selected from within the window box - not necessarily the current group). To communicate with a single process, create a new group and drag that process into it.

The *Raw Command* window will not work with playing processes and requires all processes in the chosen group to be paused.

© 2015 Allinea Software Ltd.

8 DDT: Program Input And Output

DDT collects and displays output from all processes under the *Input/Output* tab. Both standard output and error are shown, although on most MPI implementations, error is not buffered but output is and consequently can be delayed.

8.1 Viewing Standard Output And Error

Figure 67: *DDT Standard Output Window*

The Input/Output tab is at the bottom of the screen (by default).

The output may be selected and copied to the X-clipboard.

8.2 Displaying Selected Processes

You can choose whether to view the output for all processes, or just a single process.

Note: Some MPI implementations pipe stdin, stdout and stderr from every process through mpirun or rank 0.

MPI users should note that most MPI implementations place their own restrictions on program output. Some buffer it all until `MPI_Finalize` is called, others may ignore it or send it all through to one process. If your program needs to emit output as it runs, Allinea suggest writing to a file.

All users should note that many systems buffer `stdout` but not `stderr`. If you do not see your `stdout` appearing immediately, try adding an `fflush(stdout)` or equivalent to your code.

8.3 Saving Output

By right-clicking on the text it is possible to save it to a file. You also have the option to copy a selection to the clipboard.

8.4 Sending Standard Input

DDT provides an *stdin* file box in the *Run* window. This allows you to choose a file to be used as the standard input (stdin) for your program. (DDT will automatically add arguments to `mpirun`to ensure your input file is used.)

© 2015 Allinea Software Ltd.

Alternatively, you may enter the arguments directly in the *mpirun Arguments* box. For example, if using MPI directly from the *command-line* you would normally use an option to the `mpirun`such as `-stdin filename`, then you may add the same options to the *mpirun Arguments* box when starting your DDT session in the *Run* window.

It is also possible to enter input during a session. Start your program as normal, then switch to the *Input/Output* panel. Here you can see the output from your program and type input you wish to send. You may also use the *More* button to send input from a file, or send an EOF character.

Remember: Although input can be sent while your program is paused, the program must then be played to read the input and act upon it.

If you are currently viewing output for all processes then the input you type will also be sent to all processes, similarly if you are currently viewing the output for a single process then the input will be sent to just that process.

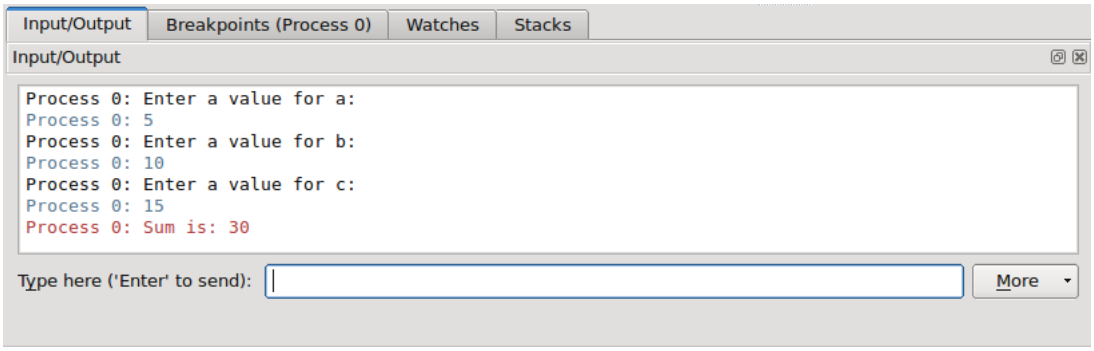


Figure 68: *DDT Sending Input*

Note: If DDT is running on a fork-based system such as Scyld, or a `-comm=shared` compiled MPICH 1, your program may not receive an EOF correctly from the input file. If your program seems to hang while waiting for the last line or byte of input, this is likely to be the problem. See the E General Troubleshooting and Known Issues or contact Allinea for a list of possible fixes.

© 2015 Allinea Software Ltd.

9 DDT: Logbook

The logbook automatically generates a log of the user's interaction with DDT, e.g. setting a breakpoint or playing the program. For each stop of the program, the reason and location is recorded together with the parallel stacks and local variables for one process.

Tracepoint values and output are logged as well.

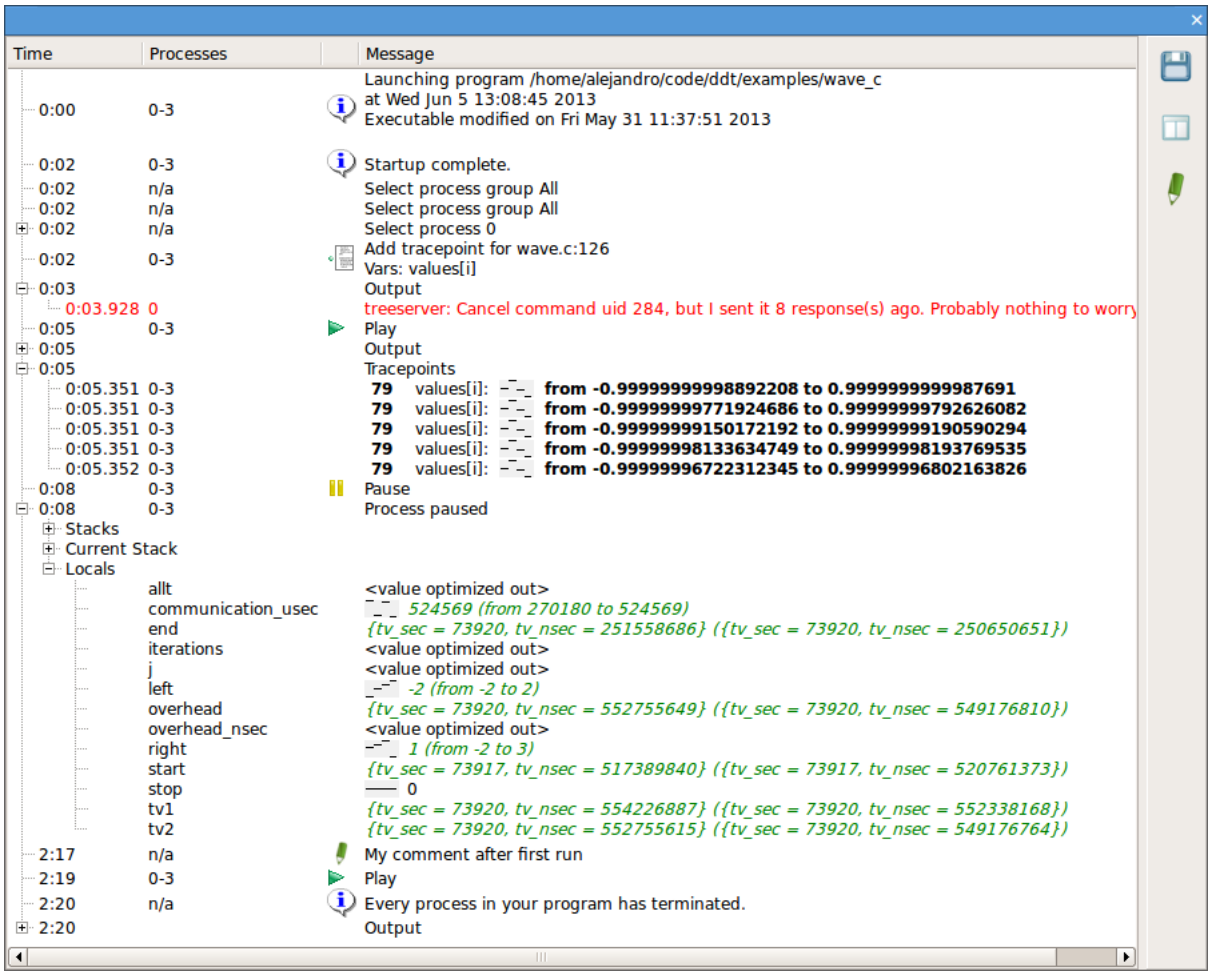

Time
Processes
Message
Launching program /home/alejandro/code/ddt/examples/wave_c
at Wed Jun 5 13:08:45 2013
Executable modified on Fri May 31 11:37:51 2013
0:00 0-3
0:02 0-3 Startup complete.
0:02 n/a Select process group All
0:02 n/a Select process group All
0:02 n/a Select process 0
0:02 0-3 Add tracepoint for wave.c:126
Vars: values[i]
0:03 Output
0:03.928 0 treeserver: Cancel command uid 284, but I sent it 8 response(s) ago. Probably nothing to worry
0:05 0-3 Play
0:05 Output
0:05 Tracepoints
0:05.351 0-3 79 values[i]: from -0.99999999998892208 to 0.9999999999987691
0:05.351 0-3 79 values[i]: from -0.99999999771924686 to 0.99999999792626082
0:05.351 0-3 79 values[i]: from -0.99999999150172192 to 0.99999999190590294
0:05.351 0-3 79 values[i]: from -0.99999998133634749 to 0.99999998193769535
0:05.352 0-3 79 values[i]: from -0.99999996722312345 to 0.99999996802163826
0:08 0-3 Pause
0:08 0-3 Process paused
Stacks
Current Stack
Locals
allt <value optimized out>
communication_usec 524569 (from 270180 to 524569)
end {tv_sec = 73920, tv_nsec = 251558686} ({tv_sec = 73920, tv_nsec = 250650651})
iterations <value optimized out>
j <value optimized out>
left -2 (from -2 to 2)
overhead {tv_sec = 73920, tv_nsec = 552755649} ({tv_sec = 73920, tv_nsec = 549176810})
overhead_nsec <value optimized out>
right 1 (from -2 to 3)
start {tv_sec = 73917, tv_nsec = 517389840} ({tv_sec = 73917, tv_nsec = 520761373})
stop 0
tv1 {tv_sec = 73920, tv_nsec = 554226887} ({tv_sec = 73920, tv_nsec = 552338168})
tv2 {tv_sec = 73920, tv_nsec = 552755615} ({tv_sec = 73920, tv_nsec = 549176764})
2:17 n/a My comment after first run
2:19 0-3 Play
2:20 n/a Every process in your program has terminated.
2:20 Output

Figure 69: *Logbook example of a debug session*

The user can export the current logbook as HTML or compare it to a previously exported one.

This enables comparative debugging and repeatability – it is always clear how a certain situation in the debugger was caused as the previous steps are visible.

9.1 Usage

The logbook is always on and does not require any additional configuration. It is integrated as Logbook tab at the bottom of the main window beside the Tracepoint Output tab.

To export the logbook click on file icon on RHS and choose a filename. A previously saved logbook can be opened using a Tools menu option.

© 2015 Allinea Software Ltd.

9.2 Comparison Window

Two logbooks can be compared side by side with the logbook comparison window. Either click the 'compare' icon on the right hand side of the Logbook View from the Tools menu or use the same icon from the Logbook tab. The current logbook can be compared with a file, or two files can be compared.

To easily spot differences the user can first align both logbooks to corresponding entries and then press the Sync button. This ensures both vertical and horizontal scrollbars of the logbooks are tied together.

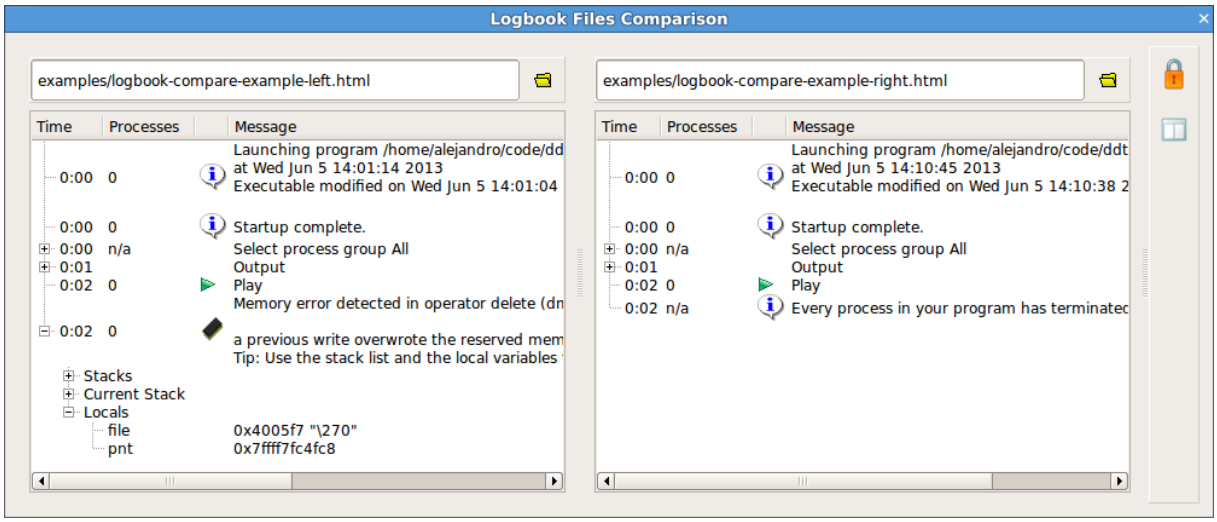


Figure 70: *Logbook comparison window with tracepoint difference selected*

© 2015 Allinea Software Ltd.

10 DDT: Message Queues

DDT's Message Queue debugging feature shows the status of the message buffers of MPI – for example showing the messages that have been sent by a process but not yet received by the target.

You can use DDT to detect common errors such as deadlock – where all processes are waiting for each other, or for detecting when messages are present that are unexpected, which can correspond to two processes disagreeing about the state of progress through a program.

This capability relies on the MPI implementation supporting this via a debugging support library: the majority of MPIs do this. Furthermore, not all implementations support the capability to the same degree, and a variance between the information provided by each implementation is to be expected.

10.1 Viewing The Message Queues

Open the *Message Queues* window by selecting *Message Queues* from the *Tools* menu. The Message Queues window will query the MPI processes for information about the state of the queues.

Whilst the window is open, you can click *Update* to refresh the current queue information. Please note that this will stop all playing processes. While DDT is gathering the data a “Please Wait” dialog may be displayed and you can cancel the request at any time.

DDT will automatically load the message queue support library from your MPI implementation (provided one exists). If it fails, an error message will be shown. Common reasons for failure to load included:

- The support library does not exist.

  Most MPIs will build the library by default, without additional configuration flags. MPICH 2 and MPICH 3 must be configured with the `--enable-debuginfo` argument. MPICH 1.2.x must be configured with the `--enable-debug` argument. Some MPIs – notably Cray's MPI – do not support message queue debugging at all.

  LAM and Open MPI automatically compile the library.

- The support library is not available on the compute nodes where the MPI processes are running.

  Please ensure the library is available – and set DDT_QUEUE_DLL if necessary to force using the library in its new location.

- The support library has moved from its original installation location.

  Please ensure the proper procedure for the MPI configuration is used – this may require you to specify the installation directory as a configuration option.

  Alternatively, you can specifically include the path to the support library in the `LD_LIBRARY_PATH`, or if this is not convenient you can set the environment variable, `DDT_QUEUE_DLL`, to the absolute pathname of the library itself (e.g. `/usr/local/mpich-1.2.7/lib/libtvmpich.so`).

- The MPI is built to a different bit-size to the debugger.

  In the unlikely case that the MPI is not built to the bit-size of the operating system, then the debugger may not be able to find a support library that is the correct size. This is unsupported.

© 2015 Allinea Software Ltd.

10.2 Interpreting the Message Queues

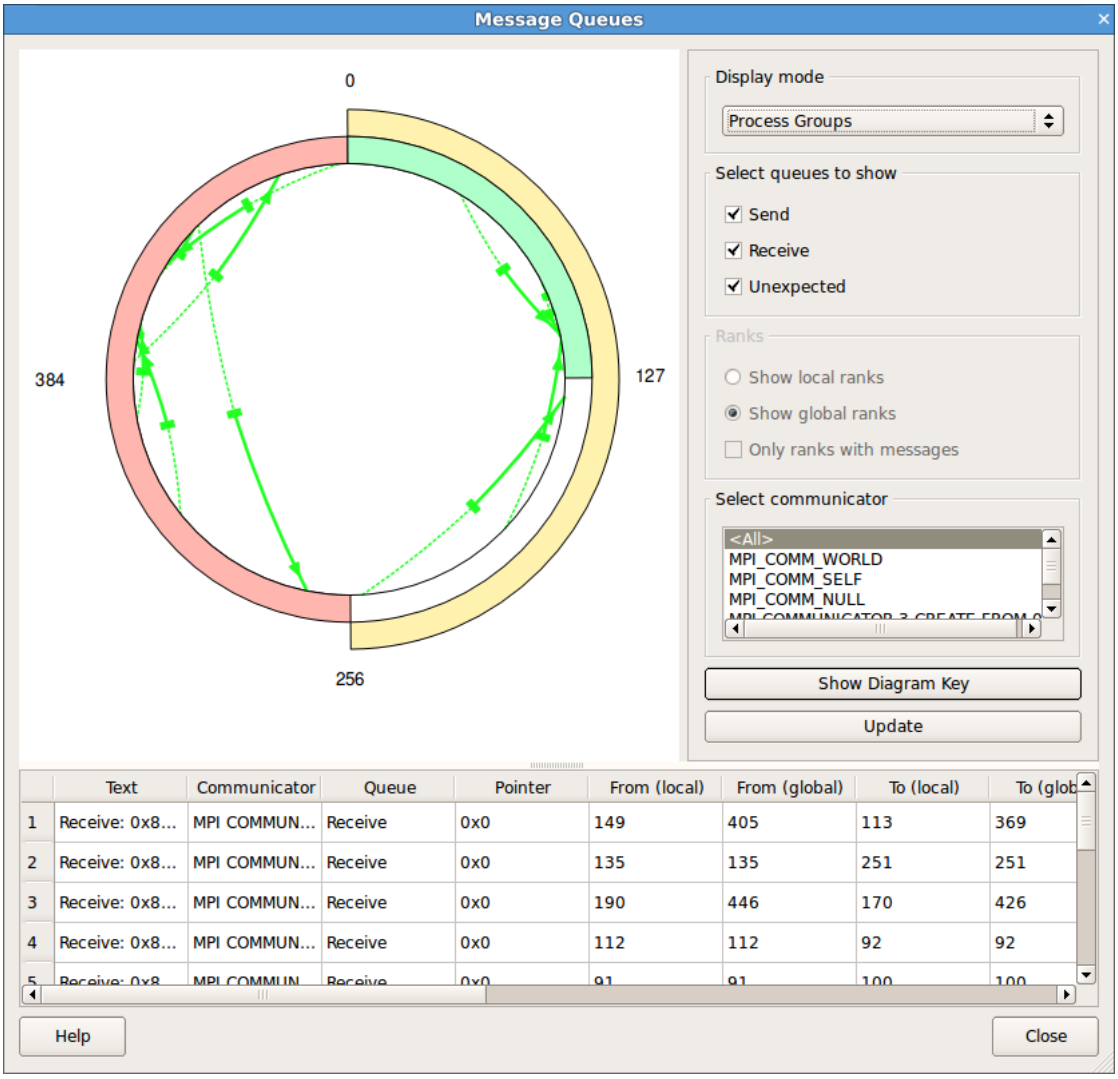


Figure 71: *Message Queue Window*

To see the messages, you must select a communicator to see the messages in that group. The ranks displayed in the diagram are the ranks within the communicator (not `MPI_COMM_WORLD`), if the *Show Local Ranks* option is selected. To see the 'usual' ranks, select *Show Global Ranks*. The messages displayed can be restricted to particular processes or groups of processes. To restrict the display in the grid to a single process, select *Individual Processes* in the *Display mode* selector, and select the rank of the process. To select a group of processes, select *Process Groups* in the *Display mode* selector and select the ring arc corresponding to the required group. Both of these display modes support multiple selections.

There are three different types of message queues about which there is information. Different colours are used to display messages from each type of queue.

Label	Description
Send Queue	Calls to MPI send functions that have not yet completed.
Receive Queue	Calls to MPI receive functions that have not yet completed.
Unexpected Message Queue	Represents messages received by the system but the corresponding receive function call has not yet been made.

© 2015 Allinea Software Ltd.

Messages in the Send queue are represented by a red arrow, pointing from the sender to the recipient. The line is solid on the sender side, but dashed on the received side (to represent a message that has been *Sent* but not yet been *Received*).

Messages in the Receive queue are represented by a green arrow, pointing from the sender to the recipient. The line is dashed on the sender side, but solid on the recipient side (to represent the recipient being ready to receive a message that has not yet been sent).

Messages in the Unexpected queue are represented by a dashed blue arrow, pointing from sender of the unexpected message to the recipient.

A message to self is indicated by a line with one end at the centre of the diagram.

Please note that the quality and availability of message queue data can vary considerably between MPI implementations – some of the data can therefore be incomplete.

10.3 Deadlock

A loop in the graph can indicate deadlock – every process waiting to receive from the preceding process in the loop.

For synchronous communications (eg. MPI_Ssend) then this is invariably a problem.

For other types of communication it can be the case (eg. with MPI_Send) that, for example, messages are 'in the ether' or in some O/S buffer and the send part of the communication is complete but the receive hasn't started. If the loop persists after playing the processes and interrupting them again, this indicates a likely deadlock.

© 2015 Allinea Software Ltd.

11 DDT: Memory Debugging

Allinea DDT has a powerful parallel memory debugging capability. This feature intercepts calls to the system memory allocation library, recording memory usage and monitoring correct usage of the library by performing heap and bounds checking.

Typical problems that can be resolved by using Allinea DDT with memory debugging enabled include:

- Memory exhaustion due to memory leaks can be prevented by examining the *Current Memory Usage* display which groups and quantifies memory according to the location at which blocks have been allocated.
- Persistent but random crashes caused by access to memory beyond the bounds of an allocation block – can be resolved by using the *Guard Pages* feature
- Crashing due to deallocation of the same memory block twice and other forms deallocation of an invalid pointers – for example deallocating a pointer that is not at the start of an allocation.

11.1 Enabling Memory Debugging

To enable memory debugging within Allinea DDT, from the *Run* window click on the *Memory Debugging* checkbox.

The default options are usually sufficient, but you may need to configure extra options (see below) if you have a multithreaded application or multithreaded MPI – such as that found on systems using Open MPI with Infiniband, or a Cray XE6 system.

With the Memory Debugging setting enabled, start your application as normal. Allinea DDT will take care of ensuring that the settings are propagated through your MPI or batch system when your application starts.

If a problem is detected and it was not possible to load the memory debugging library, a message will be displayed – and you should refer to the Configuration section in this chapter for suggested resolution steps.

Note: Memory debugging is not supported for programs that use the Xeon Phi `#pragma offload`.

11.2 CUDA Memory Debugging

Allinea DDT provides two options for debugging memory errors in CUDA programs (found in the CUDA section of the *Run* window):

When the *Track GPU allocations* option is enabled Allinea DDT tracks CUDA memory allocations *made by the host* (i.e. allocations made using functions such as `cudaMalloc`). You can find how much memory is allocated and where it was allocated from in the *Current Memory Usage window*. DDT will also detect common programming errors such as freeing memory twice.

Allocations are tracked separately for each GPU and the host (enabling *Track GPU allocations* will automatically track host-only memory allocations made using `malloc`, etc. as well). You can select between GPUs using the drop-down list in the top-right corner of the Memory Usage and Memory Statistics windows.

The *Detect invalid read/writes* option turns on the CUDA-MEMCHECK error detection tool which can detect problems such as out-of-bounds and misaligned global memory accesses.

See section 14.2 Preparing to Debug GPU Code before starting DDT.

© 2015 Allinea Software Ltd.

11.3 Configuration

Whilst configuration is often not necessary, it can be used to increase or change the memory checks and protection or to alter the information that is available. A summary of the settings is displayed on the *Run* dialog in the *Memory Debugging* section.

To examine or change the options, select the *Details* button adjacent to the *Memory Debugging* checkbox on the Run dialog, which then displays the *Memory Debugging Options* window.

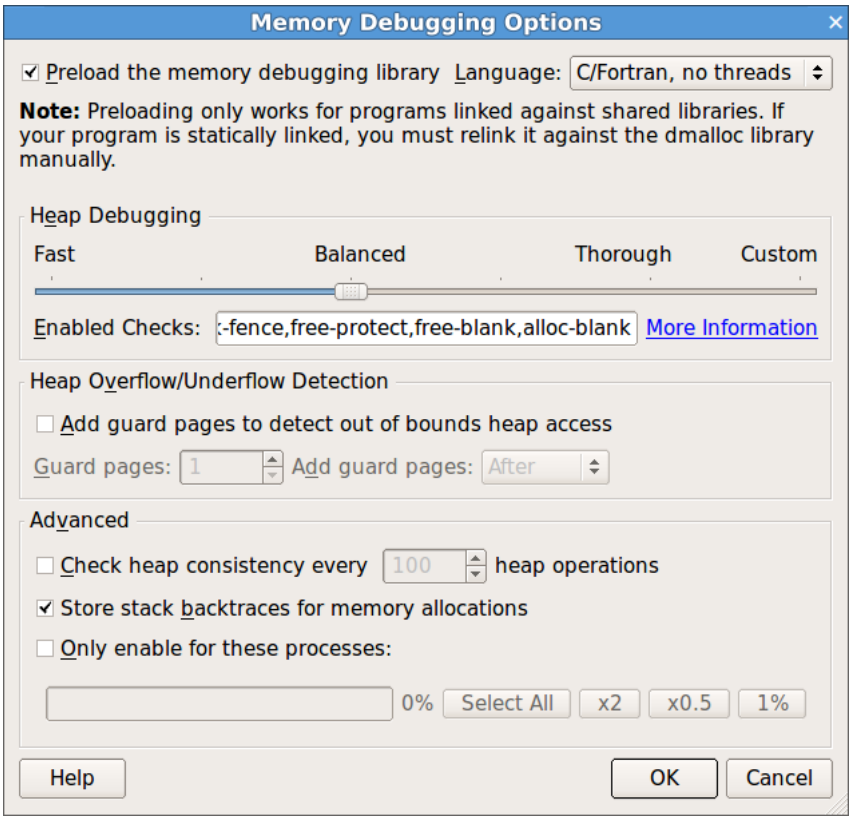


Figure 72: *Memory Debugging Options*

The two most significant options are:

1. *Preload the memory debugging library* - when this is checked, DDT will automatically load the memory debugging library. DDT can only preload the memory debugging library when you start a program through DDT **and it uses shared libraries**.

   Preloading is not possible with statically-linked programs or when attaching to a running process. See section 11.3.1 Static Linking section for more information on static linking.

   When attaching, you can set the `DMALLOC_OPTIONS` environment variable before running your program or see section 11.3.3 Changing Settings at Run Time section below.

2. The box showing *C/Fortran, No Threads* in the screen shot. You should choose the option that best matches your program – It is often sufficient to leave this set to C++/Threaded rather than continually changing this setting.

The *Heap Debugging* section allows you to trade speed for thoroughness. The two most important things to remember are:

© 2015 Allinea Software Ltd.

1. Even the fastest (leftmost) setting will catch trivial memory errors such as deallocating memory twice.
2. The further right you go, the more slowly your program will execute. In practice, the *Balanced* setting is fast enough to use and will catch almost all errors. If you come across a memory error that's difficult to pin down, choosing the *Thorough* might expose the problem earlier, but you'll need to be very patient on large, memory intensive codes (also see 11.3.3 Changing Settings at Run Time).

You can see exactly which checks are enabled for each setting in the *Enabled Checks* box. See section 11.3.2 Available Checks for a complete list of available checks.

You can turn on *Heap Overflow/Underflow Detection* to detect out of bounds heap access. See section 11.4.3 Writing Beyond An Allocated Area for more details.

Almost all users can leave the heap check interval at its default setting. It determines how often the memory debugging library will check the entire heap for consistency. This is a slow operation, so is normally performed every 100 memory allocations. This figure can be changed manually – a higher setting (1000 or above) is recommended if your program allocates and deallocates memory very frequently (e.g. inside a computation loop).

If your program runs particularly slowly with Memory Debugging enabled you may be able to get a modest speed increase by disabling the *Store backtraces for memory allocations* option. This disables stack back traces in the *View Pointer Details* and *Current Memory Usage* windows, support for custom allocators and cumulative allocation totals.

It is possible to enable Memory Debugging for only selected MPI ranks by checking the *Only enable for these processes* option and entering the ranks you want to it for.

Note: The Memory Debugging library will still be loaded into the other processes, but no errors will be reported.

Click on *OK* to save these settings, or *Cancel* to undo your changes.

Note: Choosing the wrong library to preload or the wrong number of bits may prevent DDT from starting your job, or may make memory debugging unreliable. You should check these settings if you experience problems when memory debugging is enabled.

11.3.1 Static Linking

If your program is statically linked then you must explicitly link the memory debugging library with your program in order to use the *Memory Debugging* feature in DDT.

To link with the memory debugging library you must add the appropriate flags from the table below at the **end** of the link command, but **before** every occurrence of `-lc` (if present).

Note: if in doubt use the `-ldmallocthcxx` library.

Multi-thread	C++	Bits	Library
no	no	64	`-L/path/to/ddt/lib/64 -ldmalloc -Wl,--allow-multiple-definition`
yes	no	64	`-L/path/to/ddt/lib/64 ldmallocth -Wl,--allow-multiple-definition`
no	yes	64	`-L/path/to/ddt/lib/64 -ldmallocxx -Wl,--allow-multiple-definition`

© 2015 Allinea Software Ltd.

yes	yes	64	`-L/path/to/ddt/lib/64 -ldmallocthcxx -Wl,--allow-multiple-definition`
no	no	32	`-L/path/to/ddt/lib/32 -ldmalloc -Wl,--allow-multiple-definition`
yes	no	32	`-L/path/to/ddt/lib/32 ldmallocth -Wl,--allow-multiple-definition`
no	yes	32	`-L/path/to/ddt/lib/32 -ldmallocxx -Wl,--allow-multiple-definition`
yes	yes	32	`-L/path/to/ddt/lib/32 -ldmallocthcxx -Wl,--allow-multiple-definition`

Note that `-z muldefs` is equivalent to `-Wl,--allow-multiple-definition` in the above. See section C.6 Intel Compilers and section C.8 Portland Group Compilers for compiler-specific information.

11.3.2 Available Checks

The following heap checks are available and may be enabled in the Enable Checks box:

Name	Description
`basic`	Detects invalid pointers passed to memory functions (`malloc`, `free`, `ALLOCATE`, `DEALLOCATE`, etc.)
`check-funcs`	Check the arguments of addition functions (mostly string operations) for invalid pointers.
`check-heap`	Checks for heap corruption (e.g. due to writes to invalid memory addresses).
`check-fence`	Checks the end of an allocation has not been overwritten when it is freed.
`alloc-blank`	Initialises the bytes of new allocations with a known value.
`free-blank`	Overwrites the bytes of freed memory with a known value.
`check-blank`	Check to see if space that was blanked when a pointer was allocated or when it was freed has been overwritten. Enables `alloc-blank` and `free-blank`.
`realloc-copy`	Always copy data to a new pointer when re-allocating a memory allocation (e.g. due to `realloc`)
`free-protect`	Protects freed memory where possible (using hardware memory protection) so subsequent read/writes cause a fatal error.

11.3.3 Changing Settings at Run Time

You can change most Memory Debugging settings while your program is running by selecting the `Control` → `Memory Debugging Options` menu item. In this way you can enable Memory Debugging with a minimal set of options when your program starts, set a breakpoint at a place you want to investigate for memory errors, then turn on more options when the breakpoint is hit.

© 2015 Allinea Software Ltd.

11.4 Pointer Error Detection and Validity Checking

Once you have enabled memory debugging and started debugging, all calls to the allocation and deallocation routines of heap memory will be intercepted and monitored. This allows both for automatic monitoring for errors, and for user driven inspection of pointers.

11.4.1 Library Usage Errors

If the memory debugging library reports an error Allinea DDT will display a window similar to the one shown below. This briefly reports the type of error detected and gives the option of continuing to play the program, or pausing execution.

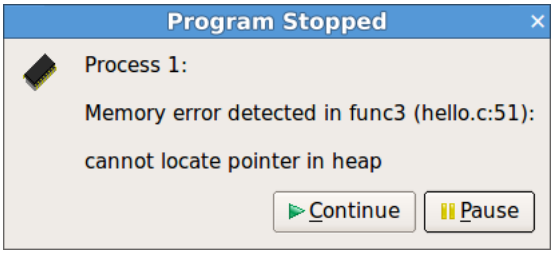


Figure 73: *Memory Error Message*

If you choose to pause the program then Allinea DDT will highlight the line of your code that was being executed when the error was reported.

Often this is enough to debug simple memory errors, such as freeing or dereferencing an unallocated variable, iterating past the end of an array and so on – as the local variables and variables on the current line will provide insight into what is happening.

If the cause of the issue is still not clear, then it is possible to examine some of the pointers referenced to see whether they're valid and which line they were allocated on, as we now explain.

11.4.2 View Pointer Details

Any of the variables or expressions in the *Evaluate* window can be right-clicked on to bring up a menu. If memory debugging is enabled, *View Pointer Details* will be available. This will display the amount of memory allocated to the pointer and which part of your code originally allocated that memory:

© 2015 Allinea Software Ltd.

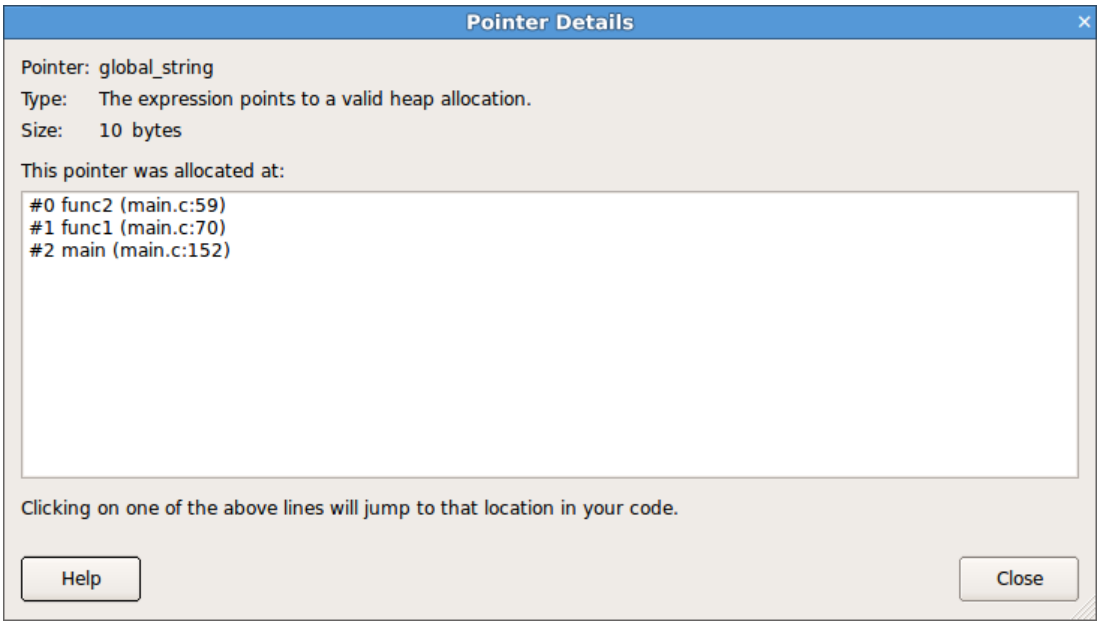

Pointer Details
Pointer: global_string
Type: The expression points to a valid heap allocation.
Size: 10 bytes
This pointer was allocated at:
#0 func2 (main.c:59)
#1 func1 (main.c:70)
#2 main (main.c:152)
Clicking on one of the above lines will jump to that location in your code.
Help
Close

Figure 74: *Pointer details*

Clicking on any of the stack frames will display the relevant section of your code, so that you can see where the variable was allocated.

Note: Only a single stack frame will be displayed if the Store stack backtraces for memory allocations option is disabled.

This feature can also be used to check the validity of heap allocated memory.

Note: Memory allocated on the heap refers to memory allocated by `malloc`, `ALLOCATE`, `new` and so on. A pointer may also point to a local variable, in which case DDT will tell you it does not point to data on the heap. This can be useful, since a common error is taking a pointer to a local variable that later goes out of scope.

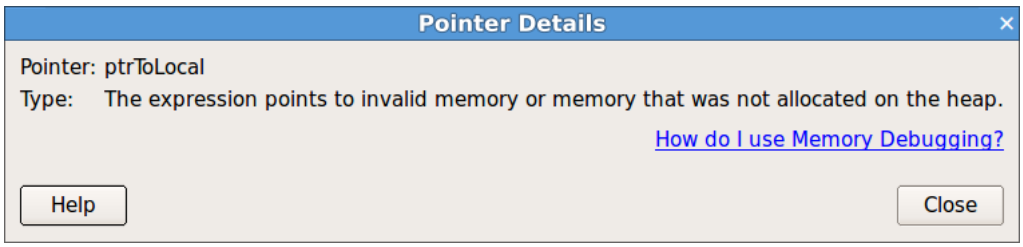

Pointer Details
Pointer: ptrToLocal
Type: The expression points to invalid memory or memory that was not allocated on the heap.
How do I use Memory Debugging?
Help
Close

Figure 75: *Invalid memory message*

This is particularly useful for checking function arguments, and key variables when things seem to be going awry. Of course, just because memory is valid doesn't mean it is the same type as you were expecting, or of the same size, or the same dimensions and so on.

Memory Type/Location

As well as invalid addresses, DDT can often tell you the type/location of the memory being pointed to. The different types are listed below.

Note: DDT may only be able to identify certain memory types with higher levels of memory debugging (see 11.3 Configuration for more information).

- Null pointer.
- Valid heap allocation.
- Fence-post area before the beginning of an allocation.
- Fence-post area beyond the end of an allocation.
- Freed heap allocation.
- Fence-post area before the beginning of a freed allocation.
- Fence-post area beyond the end a freed allocation.
- A valid GPU heap allocation.
- An address on the stack.
- The program's code section (or a shared library).
- The program's data section (or a shared library).
- The program's bss section or Fortran COMMON block (or a shared library).
- The program's executable (or a shared library).
- A memory mapped file.

For more information on fence post checking, see 11.4.4 Fencepost Checking

11.4.3 Writing Beyond An Allocated Area

Use *Heap Overflow / Underflow Detection* option to detect read or writes beyond or before an allocated block. Any attempts to read or write to the specified number of pages before or after the block will cause a segmentation violation that stops your program. Add the guard pages after the block to detect heap overflows, or before to detect heap underflows. The default value of 1 page will catch most heap overflow errors but if this doesn't work a good rule of thumb is to set the number of guard pages according to the size of a row in your largest array. The exact size of a memory page depends on your operating system but a typical size is 4Kb. So if a row of your largest array is 64Kb then set the number of pages to 64/4 = 16.

11.4.4 Fencepost Checking

DDT will also perform 'Fence Post' checking whenever the *Heap Debugging* setting is not set to *Fast*.

In this mode, an extra portion of memory is allocated at the start and/or end of your allocated block, and a pattern is written into this area. If you attempt to write beyond your data, say by a few elements, then this will be noticed by DDT, however your program will not be stopped at the exact location at which your program wrote beyond the allocated data – it will only stop at the next heap consistency check.

11.4.5 Suppressing an Error

If DDT stops at an error but you wish to ignore the error (for example, it may be in a third party library you cannot fix) then you may check the *Suppress memory errors from this line in future*. This will open the *Suppress Memory Errors* window. Here you may select which function you want to suppress errors from.

© 2015 Allinea Software Ltd.

11.5 Current Memory Usage

Memory leaks can be a significant problem for software developers. If your application's memory usage grows faster than expected or continues to grow through its execution then it is possible that memory is being allocated which is not being returned when it is no longer required.

This type of problem is typically difficult to diagnose, and particularly so in a parallel environment, but Allinea DDT is able to make this task simple.

At any point in your program you can go to *View* → *Current Memory Usage* and DDT will then display the currently allocated memory in your program for the currently selected group. For larger process groups, the processes displayed will be the ones that are using the most memory across that process group.

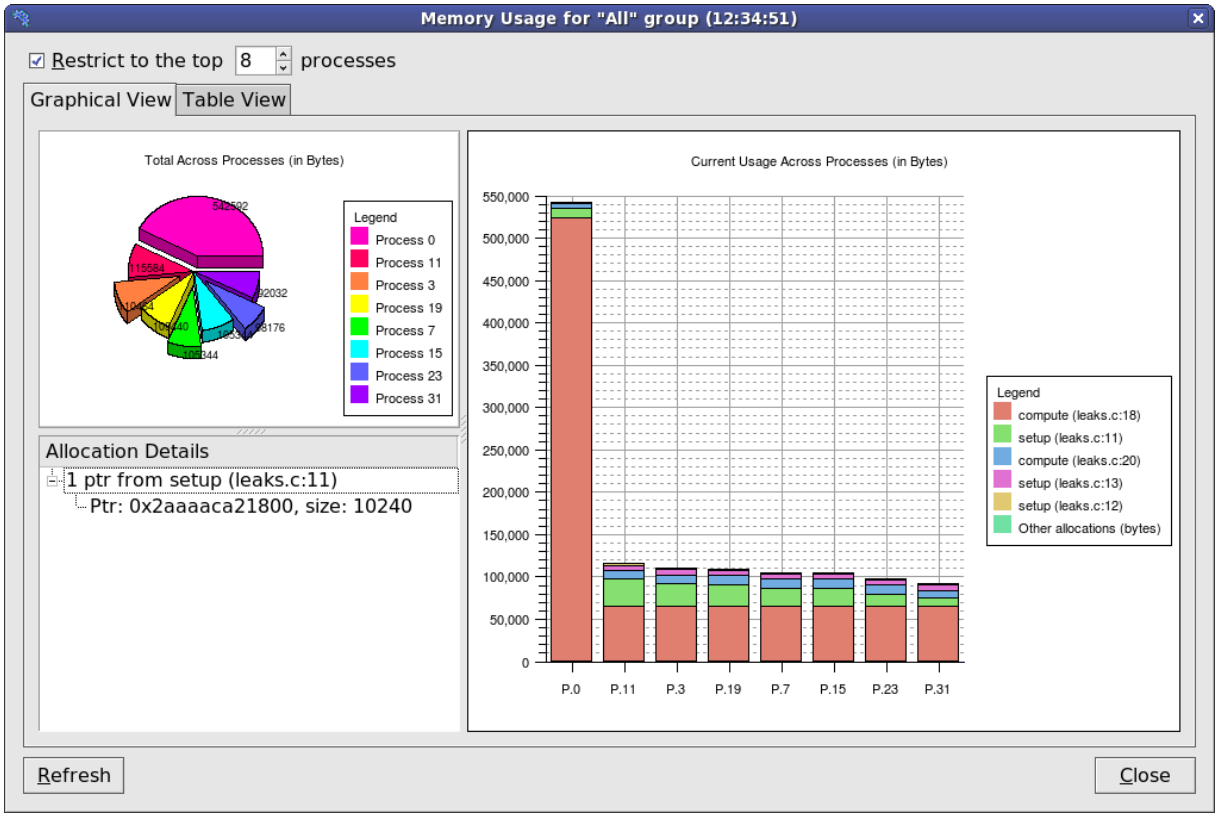


Figure 76: *Memory Usage Graphs*

The pie chart gives an at-a-glance comparison of the total memory allocated to each process. This gives an indication of the balance of memory allocations; any one process taking an unusually large amount of memory should be easily identifiable here.

The stacked bar chart on the right is where the most interesting information starts. Each process is represented by a bar, and each bar broken down into blocks of colour that represent the total amount of memory allocated by a particular function in your code. Say your program contains a loop that allocates a hundred bytes that is never freed. That's not a lot of memory. But if that loop is executed ten million times, you're looking at a gigabyte of memory being leaked! There are 6 blocks in total. The first 5 represent the 5 functions that allocated the most memory allocated, and the 6th (at the top) represents the rest of the allocated memory, wherever it is from.

As you can see, large allocations (if your program is close to the end, or these grow, then they are severe memory leaks) show up as large blocks of colour. Typically, if the memory leak does not make it into the

© 2015 Allinea Software Ltd.

top 5 allocations under any circumstances then it isn't that big a deal – although if you are still concerned you can view the data in the *Table View* yourself.

If any block of colour interests you, click on it. This will display detailed information about the memory allocations that make it up in the bottom-left pane. Scanning down this list gives you a good idea of what size allocations were made, how many and where from. Clicking on any one of these will display the 'Pointer Details' view described above, showing you exactly where that pointer was allocated from in your code.

Note: Only a single stack frame will be displayed if the Store stack backtraces for memory allocations option is disabled.

The *Table View* shows all the functions that allocated memory in your program alongside the number of allocations (*Count*) and the total number of bytes allocated (*Size*). It also shows the total memory allocated by each function's callees in the *Cumulative Count* and *Cumulative Size* columns.

For example: `func1` calls `func2` which calls `malloc` to allocate 50 bytes. DDT will report an allocation of 50 bytes against `func2` in the Size column of the *Current Memory Usage* table. DDT will also record a cumulative allocation of 50 bytes against both functions `func1` and `func2` in the *Cumulative Size* column of the table.

Another valuable use of this feature is to play the program for a while, refresh the window, play it for a bit longer, refresh the window and so on – if you pick the points at which to refresh (e.g. after units of work are complete) you can watch as the memory load of the different processes in your job fluctuates and will easily spot any areas that grow and grow – these are problematic leaks.

11.5.1 Detecting Leaks when using Custom Allocators/Memory Wrappers

Some compilers wrap memory allocations inside many other functions. In this case Allinea DDT may find, for example, that all Fortran 90 allocations are inside the same routine. This can also happen if you have written your own wrapper for memory allocation functions.

In these circumstances you will see one large block in the Current Memory Usage view. You can mark such functions as *Custom Allocators* to exclude them from the bar chart and table by right-clicking on the function and selecting the *Add Custom Allocator* menu item. Memory allocated by a custom allocator is recorded against its caller instead.

For example, if *myfunc* calls `mymalloc` and `mymalloc` is marked as a custom allocator the allocation will be recorded against `myfunc` instead. You can edit the list of custom allocators by clicking the "*Edit Custom Allocators...*" button at the bottom of the window.

11.6 Memory Statistics

The Memory Statistics view (*View* → *Memory Statistics*) shows a total of memory usage across the processes in an application. The processes using the most memory are displayed, along with the mean across all processes in the current group, which is useful for larger process counts.

© 2015 Allinea Software Ltd.

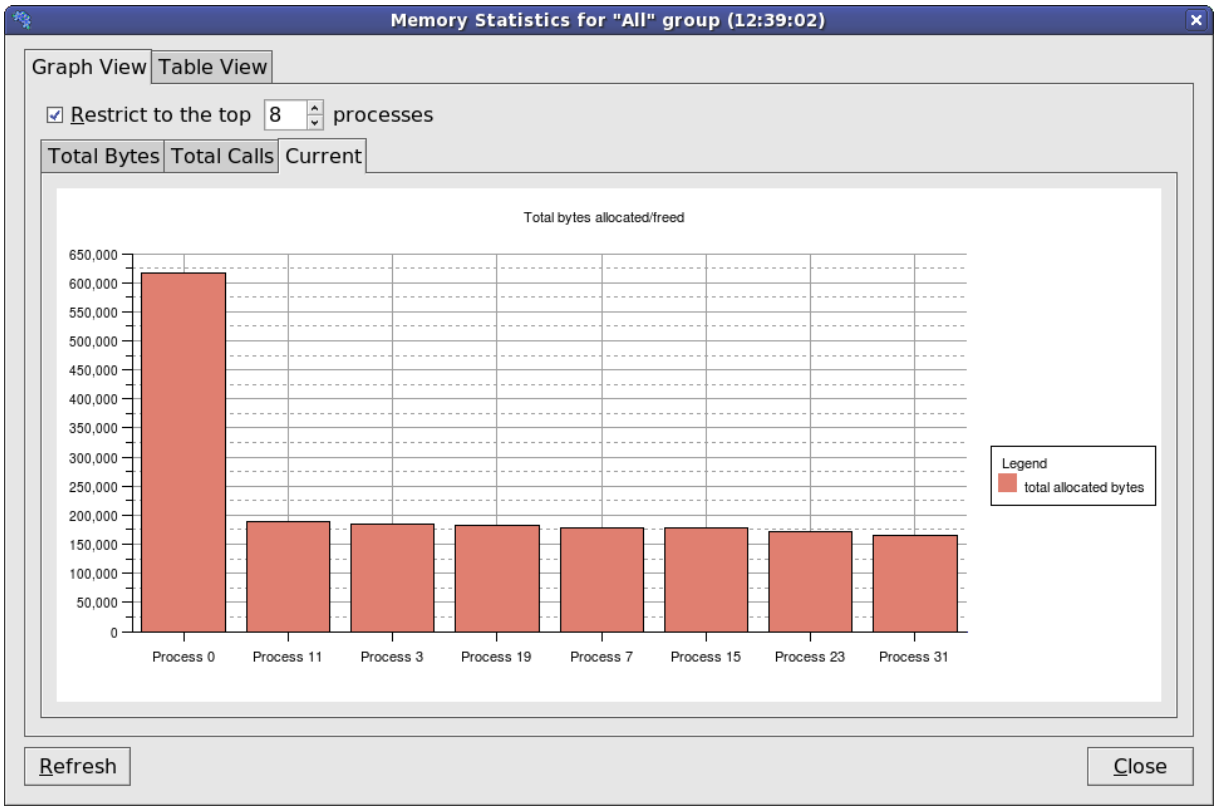


Figure 77: *Memory Statistics*

The contents and location of the memory allocations themselves are not repeated here; instead this window displays the total amount of memory allocated/freed since the program began in the left-hand pane. This can help show if your application is unbalanced, if particular processes are allocating or failing to free memory and so on. The right hand pane shows the total number of calls to allocate/free functions by process. At the end of program execution you can usually expect the total number of calls per process to be similar (depending on how your program divides up work), and memory allocation calls should always be greater than deallocation calls - anything else indicates serious problems.

© 2015 Allinea Software Ltd.

12 DDT: Checkpointing

12.1 What Is Checkpointing?

A program's entire state (or a practical subset thereof) may be recorded to memory as a checkpoint. The program may later be restored from the checkpoint and will resume execution from the recorded state.

Sometimes you are not sure what information you need to diagnose a bug until it is too late to get it. For example, a program may crash because a variable has a particular unexpected value. You want to know where the variable was set to that value but it is too late to set a watch on it. However if you have an earlier checkpoint of the program you can restore the checkpoint, set the watch, and then let it fail again.

Checkpoints in DDT are stored in memory. They are valid for the life time of a session but are lost when the session is ended.

12.2 How To Checkpoint

To checkpoint your program, click the *Checkpoint* button on the tool bar . The first time you click the button you will be asked to select a checkpoint provider. If no checkpoint providers support the current MPI and debugger an error message will be displayed instead.

When the checkpoint has completed a new window will open displaying the name of the new checkpoint.

12.3 Restoring A Checkpoint

To restore a checkpoint, click the *Restore Checkpoint* button on the tool bar . A new window will open with a list of available checkpoints. Select a checkpoint then click the Ok button.

The program state will be restored to the checkpoint. The Parallel Stack View, Locals View, etc. will all be updated with the new program state.

© 2015 Allinea Software Ltd.

13 DDT: Using and Writing Plugins

Plugins are a quick and easy way to preload a library into your application and define some breakpoints and tracepoints during its use. They consist of an XML file which instructs DDT what to do and where to set breakpoints or tracepoints.

Examples are MPI correctness checking libraries, or you could also define a library that is preloaded with your application that could perform your own monitoring of the application. It also enables a message to be displayed to the user when breakpoints are hit, displaying, for example, an error message where the message is provided by the library in a variable.

13.1 Supported Plugins

DDT supports plugins for two MPI correctness-checking libraries:

- Intel Message Checker, part of the Intel Trace Analyser and Collector (Commercial with free evaluation: http://software.intel.com/en-us/intel-trace-analyzer/) version 7.1
- Marmot (Open source: http://www.hlrs.de/organization/amt/projects/marmot), support expected in version 2.2 and above.

13.2 Installing a Plugin

To install a plugin, locate the XML DDT plugin file provided by your application vendor and copy it to:

```
{allinea-tools installation directory}/plugins/
```

It will then appear in DDT's list of available plugins on the DDT - Run dialog.

Each plugin takes the form of an XML file in this directory. These files are usually provided by third-party vendors to enable their application to integrate with DDT. A plugin for the Intel Message Checker (part of the Intel Trace Analyser and Collector) is included with the DDT distribution.

13.3 Using a Plugin

To activate a plugin in DDT, simply click on the checkbox next to it in the window, then run your application. Plugins may automatically perform one or more of the following actions:

- Load a particular dynamic library into your program
- Pause your program and show a message when a certain event such as a warning or error occurs
- Start extra, optionally hidden MPI processes (see the Writing Plugins section for more details on this).
- Set tracepoints – which log the variables during an execution.

If DDT says it cannot load one of the plugins you have selected, check that the application is correctly installed, and that the paths inside the XML plugin file match the installation path of the application. Example Plugin: MPI History Library

DDT's `plugin` directory contains a small set of files that make a plugin to log MPI communication.

- `Makefile` – Builds the library and the configuration file for the plugin.

© 2015 Allinea Software Ltd.

- `README.wrapper` – Details the installation, usage and limitations
- `wrapper-config` – Used to create the plugin XML config file - used by DDT to preload the library and set tracepoints which will log the correct variables.
- `wrapper-source` – Used to automatically generate the source code for the library which will wrap the original MPI calls.

The plugin is designed to wrap around many of the core MPI functions and seamlessly intercept calls to log information which is then displayed in DDT. It is targeted at MPI implementations which use dynamic linking, as this can be supported without relinking the debugged application.

Static MPI implementations can be made to work also, but this is outside the scope of this version.

This package must be compiled before first use - in order to be compatible with your MPI version. It will not appear in DDT's GUI until this is done.

To install - as a non-root user - in your local `~/.allinea/plugins` directory.

```
make local
```

To install - as root in the DDT `plugins` directory

```
make
```

Once you have run the above, start DDT and to enable the plugin, click the *Details...* button to expand the *Plugins* section of the *Run* window. Select `History v1.0`, and start your job as normal. DDT will take care of preloading the library and setting default tracepoints.

This plugin records call counts, total sent byte counts, and the arguments used in MPI function calls. Function calls and arguments are displayed (in blue) in the Input/Output panel.

The function counts are available -- in the form of a variable

`_MPIHistoryCount_{function}`

The sent bytes counters are accumulated for most functions - but specifically they are not added for the vector operations such as `MPI_Gatherv`. These count variables within the processes are available for use within DDT - in components such as the cross-process comparison window, enabling a check that - say - the count of MPI_Barriers is consistent, or primitive MPI bytes sent profiling information to be discovered.

The library does not record the received bytes - as most MPI receive calls in isolation only contain a maximum number of bytes allowed, rather than bytes received. The MPI status is logged, the `SOURCE` tag therein enables the sending process to be identified.

There is no per-communicator logging in this version.

This version is for demonstration purposes for the tracepoints and plugin features. It could generate excessive logged information, or cause your application to run slowly if it is a heavy communicator. This library can be easily extended – or its logging can be reduced by removing the tracepoints from the generated `history.xml` file (stored in `ALLINEA_TOOLS_PATH` or `~/.allinea/plugins`) – which would make execution considerably faster, but still retain the byte and function counts for the MPI functions.

13.4 Writing a Plugin

Writing a plugin for DDT is easy. All that is needed is an XML plugin file that looks something like this:

```
<plugin name="Sample v1.0" description="A sample plugin that
   demonstrates DDT's plugin interface.">
   <preload name="samplelib1" />
   <preload name="samplelib2" />
   <environment name="SUPPRESS_LOG" value="1" />
   <environment name="ANOTHER_VAR" value="some value" />
   <breakpoint location="sample_log" action="log" message_variable
      ="message" />
   <breakpoint location="sample_err" action="message_box"
      message_variable="message" />
   <extra_control_process hide="last" />
</plugin>
```

Only the surrounding `plugin` tag is required – all the other tags are entirely optional. A complete description of each appears in the table below. If you are interested in providing a plugin for DDT as part of your application bundle, we will be happy to provide you with any assistance you need getting up and running. Contact support@allinea.com for more information.

13.5 Plugin Reference

Tag	Attribute	Description
`plugin`	`name`	The plugin's unique name. This should include the application/library the plugin is for, and its version. This is shown in the *DDT – Run* dialog.
`plugin`	`description`	A short snippet of text to describe the purpose of the plugin/application to the user. This is also shown in the *DDT – Run* dialog.
`preload`	`name`	Instructs DDT to preload a shared library of this name into the user's application. The shared library must be locatable using `LD_LIBRARY_PATH`, or the OS will not be able to load it.
`environment`	`name`	Instructs DDT to set a particular environment variable before running the user's application.
`environment`	`value`	The value that this environment variable should be set to.
`breakpoint`	`location`	Instructs DDT to add a breakpoint at this location in the code. The location may be in a preloaded shared library (see above). Typically this will be a function name, or a fully-qualified C++ namespace and class name. C++ class members must include their signature and be enclosed in single quotes, e.g. `'MyNamespace::DebugServer::breakpointOnError(char*)'`
`breakpoint`	`action`	Only `message_box` is supported in this release. Other settings will cause DDT to stop at the breakpoint but take no action.

© 2015 Allinea Software Ltd.

`breakpoint`	`message_variable`	A `char*` or `const char*` variable that contains a message to be shown to the user. DDT will group identical messages from different processes together before displaying them to the user in a message box.
`extra_control_process`	`hide`	Instructs DDT to start one more MPI process than the user requested. The optional `hide` attribute can be first or last, and will cause DDT to hide the `first` or `last` process in `MPI_COMM_WORLD` from the user. This process will be allowed to execute whenever at least one other MPI process is executing, and messages or breakpoints (see above) occurring in this process will appear to come from all processes at once. This is only necessary for tools such as Marmot that use an extra MPI process to perform various runtime checks on the rest of the MPI program.
`tracepoint`	`location`	See breakpoint location.
`tracepoint`	`variables`	A comma-separated list of variables to log on every passing of the tracepoint location.

© 2015 Allinea Software Ltd.

14 DDT: CUDA GPU Debugging

MAP does not support CUDA in this release.

Allinea DDT is able to debug applications that use NVIDIA CUDA, with actual debugging of the code running on the GPU, simultaneously whilst debugging the host CPU code.

Allinea supports a number of GPU compilers:

- NVCC – the NVIDIA compilers
- CAPS HMPP – on-device and host debugging support for breakpoints and viewing of some on-device variables with CAPS 3.2 release.
- Cray OpenACC – full debugging support for on-device and host code debugging.
- PGI CUDA Fortran and the PGI Accelerator Model can be debugged but only running on the host (CPU). Please see the C.8 Portland Group Compilers for further details.

The CUDA toolkits and their drivers for toolkits version 4.0 and above are supported by Allinea DDT.

14.1 Licensing

In order to debug CUDA programs with Allinea DDT, a CUDA-enabled licence key is required, which is an additional option to default licences. If CUDA is not included with a licence, the CUDA options will be greyed-out on the run dialog of DDT.

Whilst debugging a CUDA program, an additional process from your licence is used for each GPU. An exception to this is that single process licences will still allow the debugging of a single GPU.

Please note that in order to serve a *floating* CUDA licence, you will need to use the licence server shipped with DDT 2.6 or later.

14.2 Preparing to Debug GPU Code

In order to debug your GPU program, you may need to add additional compiler command line options to enable GPU debugging.

For NVIDIA's `nvcc` compiler, kernels must be compiled with the "`-g -G`" flags. This enables generation of information for debuggers in the kernels, and will also disable some optimisations that would hinder debugging. To use memory debugging in DDT with CUDA 5.5 or later "`--cudart shared`" must also be passed to `nvcc`.

For other compilers, please refer to 14.9 GPU Language Support of this guide and C Compiler Notes and Known Issues and your vendor's own documentation.

Note: At this point OpenCL debugging of GPUs is not supported.

14.3 Launching the Application

To launch a CUDA job, tick the CUDA box on the run dialog before clicking run/submit. You may also enable memory debugging for CUDA programs from the CUDA section, see section 11.2 CUDA Memory Debugging for details.

© 2015 Allinea Software Ltd.

Attaching to running CUDA applications is not possible if the application has already initialized the driver, for example having executed any kernel or called any functions from the CUDA library. For MPI applications it is essential to place all CUDA initialization after the `MPI_Init` call.

14.4 Controlling GPU threads

Controlling GPU threads is integrated with the standard DDT controls – so that the usual play, pause, and breakpoints are all applicable to GPU kernels, for example.

As GPUs have different execution models to CPUs, there are some behavioural differences that we now detail.

14.4.1 Breakpoints

CUDA Breakpoints can be set in the same manner as other breakpoints in DDT (See section 6.7 Setting Breakpoints).

Breakpoints affect all GPU threads, and cause the application to stop whenever a thread reaches the breakpoint. Where kernels have similar workload across blocks and grids, then threads will tend to reach the breakpoint together and the kernel will pause once per set of blocks that are scheduled (ie. set of threads that fit on the GPU at any one time). Where kernels have divergent distributions of work across threads, then timing may be such that threads within a running kernel will hit a breakpoint and pause the kernel – and after subsequently continuining, more threads within the currently scheduled set of blocks will hit the breakpoint and pause the application again.

In order to apply breakpoints to individual blocks, warps or threads, conditional breakpoints can be used – for example using the built-in variables threadIdx.x (and threadIdx.y or threadIdx.z as appropriate) for thread indexes and setting the condition appropriately.

Where a kernel pauses at a breakpoint, the currently selected GPU thread will be changed if the previously selected thread is no longer “alive”.

14.4.2 Stepping

The GPU execution model is noticeably different from that of the host CPU. In the context of stepping operations – ie. step in, step over or step out – there are critical differences to note.

The smallest execution unit on a GPU is a warp – which on current NVIDIA GPUs is 32 threads. Step operations can operate on warps but nothing smaller.

Allinea DDT also makes it possible to step whole blocks, whole kernels or whole devices. The stepping mode is selected using the drop down list in the CUDA Thread Selector.

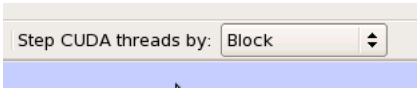


Figure 78: *Selection of GPU Stepping Mode*

Note: GPU execution under the control of a debugger is not as fast as running without a debugger. When stepping blocks and kernels these are sequentialized into warps and hence stepping of units larger than a warp may be slow. It is not unusual for a step operation to take 60 seconds on a large kernel, particularly on newer devices where a step could involve stepping over a function call.

© 2015 Allinea Software Ltd.

It is not currently possible to “step over” or “step out” of inlined GPU functions.

Note: GPU functions are often inlined by the compiler. This can be avoided (dependent on hardware) by specifying the `__noinline__` keyword in your function declaration, and by compiling your code for a later GPU profile. e.g. by adding `-arch=sm_20` to your compile line.

14.4.3 Running and Pausing

Clicking the “Play/Continue” button in DDT will run all GPU threads. It is not possible to run individual blocks, warps or threads.

The pause button will pause a running kernel, although it should be noted that the pause operation is not as quick for GPUs as for regular CPUs.

14.5 Examining GPU Threads and Data

Much of the user interface when working with GPUs is unchanged from regular MPI or multithreaded debugging. However, there are a number of enhancements and additional features that have been added to help understand the state of GPU applications.

These changes are summarised in this section.

14.5.1 Selecting GPU Threads

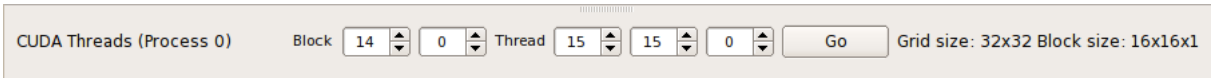


Figure 79: *GPU Thread Selector*

The Thread Selector allows you to select your current GPU thread. The current thread is used for the variable evaluation windows in DDT, along with the various GPU stepping operations.

The first entries represent the block index, and the subsequent entries represent the 3D thread index inside that block.

Changing the current thread will update the local variables, the evaluations, and the current line displays and source code displays to reflect the change.

The thread selector is also updated to display the current GPU thread if it changes as a result of any other operation – for example if:

- The user changes threads by selecting an item in the Parallel Stack View
- A memory error is detected and is attributed to a particular thread
- The kernel has progressed, and the previously selected thread is no longer present in the device

The GPU Thread Selector also displays the dimensions of the grid and blocks in your program.

It is only possible to inspect/control threads in the set of blocks that are actually loaded in to the GPU. If you try to select a thread that is not currently loaded, a message will be displayed.

Note: The thread selector is only displayed when there is a GPU kernel active.

© 2015 Allinea Software Ltd.

14.5.2 Viewing GPU Thread Locations

The Parallel Stack View has been updated to display the location and number of GPU threads.

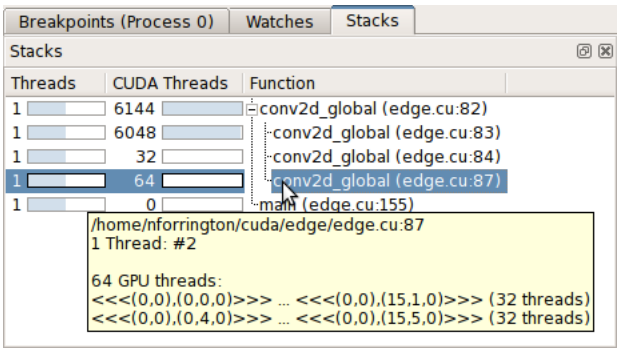


Figure 80: *CUDA threads in the parallel stack view*

Clicking an item in the Parallel Stack View will select the appropriate GPU thread – updating the variable display components accordingly and moving the source code viewer to the appropriate location.

Hovering over an item in the Parallel Stack view will also allow you to see which individual GPU thread ranges are at a location, as well as the size of each range.

14.5.3 Understanding Kernel Progress

Given a simple kernel that is to calculate an output value for each index in an array, it is not easy to check whether the value at position x in an array has been calculated, or whether the calculating thread has yet to be scheduled.

This contrasts sharply with scalar programming, where if the counter of a (up-)loop exceeds x then the value of index x can be taken as being the final value. If it is difficult to decide whether array data is fresh or stale, then clearly this will be a major issue during debugging.

Allinea DDT includes a component that makes this easy – the Kernel Progress display, which will appear at the bottom of the user interface by default when a kernel is in progress.

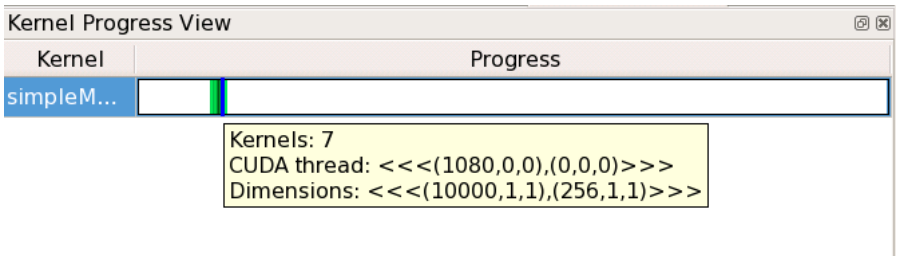


Figure 81: *Kernel Progress Display*

This view identifies the kernels that are in progress – with the number of kernels identified and grouped by different kernel identifiers (ie. kernel name) across processes – and using a coloured progress bar to identify which GPU threads are in progress. The progress bar is a projection onto a straight line of the (potentially) 6-dimensional GPU block and thread indexing system and is tailored to the sizes of the kernels operating in the application.

© 2015 Allinea Software Ltd.

By clicking within the colour highlighted sections of this progress bar, a GPU thread will be selected that matches the click location as closely *as* possible. Selected GPU threads are coloured blue. For deselected GPU threads, the ones that are scheduled are coloured green whereas the unscheduled ones are white.

14.5.4 Source Code Viewer

The source code viewer allows you to visualise the program flow through your source code by highlighting lines in the current stack trace. When debugging GPU kernels, it will colour highlight lines with GPU threads present and display the GPU threads in a similar manner to that of regular CPU threads and processes. Hovering over a highlighted line in the code viewer will display a summary of the GPU threads on that line.

14.6 GPU Devices Information

One of the challenges of GPU programming is in discovering device parameters, such as the number of registers or the device type, and whether a device is present.

In order to assist in this, Allinea DDT includes a GPU Devices display. This display examines the GPUs that are present and in use across an application, and groups the information together scalably for multi-process systems.

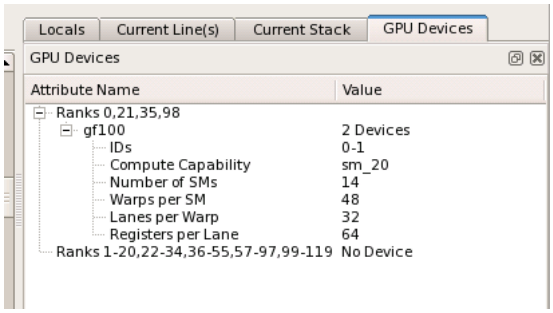


Figure 82: *GPU Devices*

Note: For CUDA 4.0, devices are only listed after initialization.

14.7 Attaching to running GPU applications

Attaching to a running GPU application and then debugging the GPU threads is only supported for

- CUDA 5 and above versions of the toolkit,
- Fermi class cards and their successors. This includes Tesla C2050/2070, K10, and K20.

To attach to a running job, please see the Section 4.8 Attaching To Running Programs – and select the *Debug CUDA* button on the attach window.

© 2015 Allinea Software Ltd.

14.8 Known Issues / Limitations

14.8.1 Debugging Multiple GPU processes (CUDA 4.0 and below)

- With CUDA 4.0, Allinea DDT can only debug a single GPU process per host. When trying to debug multiple processes with CUDA support enabled, DDT will disable GPU debugging for all but one process.

  In order to debug multiple processes with GPU support you must configure your job launch mechanism to launch each process on a separate host.

- When debugging multiple GPU processes on the same machine, pausing the process with GPU debugging enabled may also pause kernels launched by other processes.

14.8.2 Using Multiple GPU processes (CUDA 4.1 and above)

CUDA 4.1 allows debugging of multiple CUDA processes on the same node. However, each process will still attempt to reserve all of the available GPUs for debugging.

This works for the case where a single process debugs all GPUs on a node, but not for multiple processes debugging a single GPU.

A temporary workaround when using Open MPI is to export the following environment variable before starting DDT:

```
DDT_CUDA_DEVICE_VAR=OMPI_COMM_WORLD_LOCAL_RANK
```

This will assign a single device (based on local rank) to each process. In addition:

- You must have *Open MPI (Compatibility)* selected in the *File* → *Options* (*DDT* → *Preferences* on Mac OS X) . (Not Open MPI).
- The device selected for each process will be the only device visible when enumerating GPUs. This cause manual GPU selection code to stop working (due to changing device IDs, etc).

14.8.3 Thread control

The focus on thread feature in DDT isn't supported, as it can lock up the GPU. This means that it is not currently possible to control multiple GPUs in the same process individually.

14.8.4 General

- DDT supports versions 4.0 onwards of the NVIDIA CUDA toolkit. In all cases, the most recent CUDA toolkit and driver versions is recommended.
- X11 cannot be running on any GPU used for debugging. (Any GPU running X11 will be excluded from device enumeration.)
- You must compile with `-g -G` to enable GPU debugging – otherwise your program will run through the contents of kernels without stopping.
- Debugging 32-bit CUDA code on a 64-bit host system is not supported.

© 2015 Allinea Software Ltd.

- It is not yet possible to spot unsuccessful kernel launches or failures. An error code is provided by `getCudaLastError()` in the SDK which you can call in your code to detect this. Currently the debugger cannot check this without resetting it, which is not desirable behaviour.
- Device memory allocated via `cudaMalloc()` is not visible outside of the kernel function.
- Not all illegal program behaviour can be caught in the debugger - e.g. divide-by-zero.
- Device allocations larger than 100 MB on Tesla GPUs, and larger than 32 MB on Fermi GPUs, may not be accessible.
- Breakpoints in divergent code may not behave as expected.
- Debugging applications with multiple CUDA contexts running on the same GPU is not supported.
- If CUDA environment variable `CUDA_VISIBLE_DEVICES` <index> is used to target a particular GPU, then make sure no X server is running on any of the GPUs. (Also note that any GPU running X will be excluded from enumeration, with may affect the device Ids).
- The CUDA 5 driver requires that applications be debugged in a CUDA 5 mode – if your system is running the CUDA 5 driver and using the CUDA 4.x toolkit, you should force DDT to use CUDA 5 mode by setting `DDT_FORCE_CUDA_VERSION=5` – or consider upgrading to CUDA 5 toolkit.
- If memory debugging and CUDA support are enabled in DDT then only threaded memory preloads are available.

14.8.5 Pre sm_20 GPUs

For GPUs that have SM type less than `sm_20` (or when code is compiled targeting SM type less than `sm_20`), the following issues may apply.

- GPU code targeting less than SM type `sm_20` will inline all function calls. This can lead to behaviour such as not being able to step over/out of subroutines.
- Debugging applications using textures is not supported on GPUs with SM type less than `sm_20`.
- If you are debugging code in device functions that get called by multiple kernels, then setting a breakpoint in the device function will insert the breakpoint in only one of the kernels.

14.8.6 Workaround for unsupported gcc versions with nvcc

CUDA toolkit 4.0 doesn't support gcc/g++ versions onwards 4.4. An example of how to workaround this (for Ubuntu systems) is provided below:

Install an older version of gcc/g++. e.g.:

```
sudo apt-get install g++-4.4
```

Create a new bin directory and symlink the compilers in to it:

```
sudo mkdir /usr/local/gcc-44
sudo ln /usr/bin/gcc-4.4 /usr/local/gcc-44/gcc
sudo ln /usr/bin/g++-4.4 /usr/local/gcc-44/g++
```

Tell `nvcc` to use the new bin directory by editing the `nvcc.profile` file in the same directory as `nvcc` and appending the following line:

```
compiler-bindir = /usr/local/gcc-44
```

`nvcc` will now use the older supported compiler version.

14.8.7 Debugging Multiple GPU processes on Cray limitations

It is not possible to debug multiple CUDA processes on a single node on a Cray machine, you must run with 1 process per node.

14.9 GPU Language Support

In addition to the native nvcc compiler, a number of other compilers are supported.

At this point in time, debugging of OpenCL is not supported on the device.

14.9.1 CAPS HMPP

CAPS HMPP 3.2 code can be debugged in both the host and the GPU. The *Stop on kernel launch* feature can be used to identify the launch points of HMPP codelets, and breakpoints inside CAPS HMPP codelets can also be directly inserted.

The required compilation options and environment variables to the compiler are:

```
hmpp g -f -k ifort basic.f90 -o basic
```

Stepping inside a GPU codelet or pausing is supported with CAPS HMPP 3.0 and above.

The host code can be debugged as normal, subject to providing `-g` flags to the compilers as normal.

DDT will syntax highlight HMPP pragmas detected in the source.

F90 arrays and expressions are correctly displayed by DDT, including multi-dimensional arrays and arrays with (eg.) negative lower bounds.

Known issue: Local loop index variables may be incorrect, depending on the loop structure – notably nested loops may be flattened to a single loop and the mapping to a code's original index variables is not currently available.

14.9.2 Cray OpenACC

Cray OpenMP Accelerator Extensions are fully supported by Allinea DDT. Code pragmas are highlighted, most variables are visible within the device, and stepping and breakpoints in the GPU code is supported. The compiler flag `-g` is required for enabling device (GPU-based) debugging – `-O0` should not be used, as this disables using of the device and runs the accelerated regions on the CPU.

Known issue: Pointers in accelerator code cannot be dereferenced in CCE 8.0.

Known issue: Memory consumption in debugging mode can be considerably higher than regular mode, if issues with memory exhaustion arise, consider using the environment variable `CRAY_ACC_MALLOC_HEAPSIZE` to set total heap size (bytes) used on the device, which can make more memory available to the application.

© 2015 Allinea Software Ltd.

14.9.3 PGI Accelerators and CUDA Fortran

PGI Accelerator applications can be debugged when running on the host processor. To debug accelerator code it is recommended to target execution on the host process only – with the `-ta=host` compiler flag.

PGI CUDA Fortran can be debugged using Allinea DDT, by adding the `-Mcuda=emu` flag to the compiler. In this case, the CUDA will also run on the host CPU, and will use as many threads as there are host cores, allowing many thread issues to be detected.

Known issue: debugging inside the GPU is not supported – for both models, debugging is perfomed on CPU versions of the GPU code.

© 2015 Allinea Software Ltd.

15 DDT: Offline Debugging

Offline debugging is a mode of running Allinea DDT in which an application is run, under the control of the debugger, but without user intervention and without a user interface.

There are many situations where running under this scenario will be useful, for example when access to a machine is not immediately available and may not be available during the working day. The application can run with features such as tracepoints and memory debugging enabled, and will produce a report at the end of the execution.

15.1 Using Offline Debugging

To launch DDT in this mode, the `-offline` argument and a filename is specified. A filename with a `.html` extension will cause a HTML version of the output to be produced, in other cases a plain text report is generated.

```
ddt -offline myjob.html -n 4 myapplication arg1 arg2
```

or

```
ddt -offline myjob.txt -n 4 myapplication arg1 arg2
```

Additional arguments can be used to set breakpoints (at which the stack of the stopping processes will be recorded before they are continued), or to set tracepoints at which variable values will be recorded.

Settings from your current DDT configuration file will be taken, unless over-ridden on the command line.

Command line options that are of the most significance for this mode of running are:

- `-ddtsession` *`sessionfile`* - run in offline mode using settings saved using the Save Session option from DDT's File menu.
- `-n` *`nnn`* - run with *`nnn`* processes
- `-memory` enable memory debugging
- `-trace-at LOCATION[,N:M:P],VAR1,VAR2,...` - set a tracepoint at location, beginning recording after the `N`'th visit of each process to the location, and recording every `M`'th subsequent pass until it has been triggered `P` times. Record the value of variable `VAR1`, `VAR2`.

  Example:

  ```
  main.c:22,-:2:-,x
  ```

  will record x every 2nd passage of line 22.
- `-break-at LOCATION[,N:M:P]` - set a breakpoint at `LOCATION` (either `file:line` or `function`), optionally starting after the `N`'th pass, triggering every `M` passes and stopping after it has been triggered `P` times.
- The stack traces of pausing processes will be recorded, before the processes are then made to continue, and will contain the variables of one of the processes in the set of processes that have paused.

  Examples:

  ```
  -break-at main
  -break-at main.c:22
  ```

© 2015 Allinea Software Ltd.

The application will run to completion, or to the end of the job.

When errors occur, for example an application crash, the stack back trace of crashing processes will be recorded to the offline output file. In offline mode, DDT will always act as if the user had clicked Continue if the continue option was available in an equivalent “online” debugging session.

15.2 Offline Report Output (HTML)

The output file is broken into three sections, *Messages*, *Tracepoints*, and *Output* and written when debugging has finished.

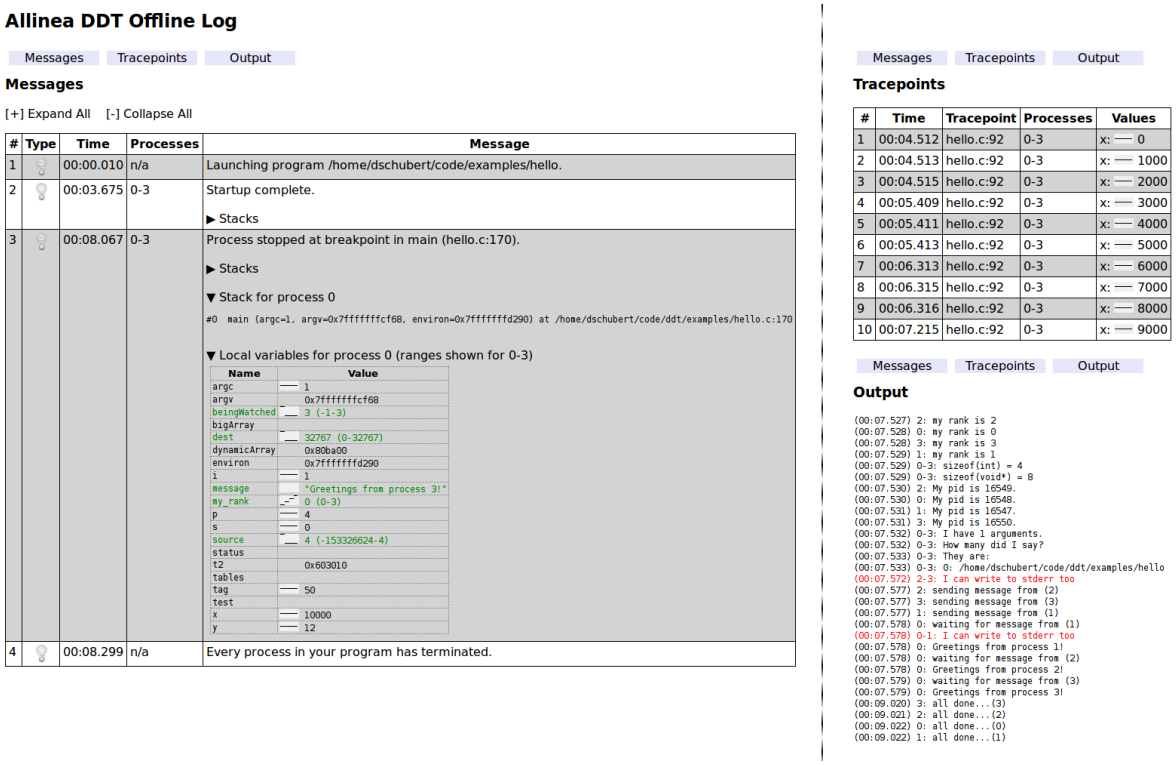

Allinea DDT Offline Log
Messages
Tracepoints
Output
Messages
[+] Expand All
[-] Collapse All
Type Time Processes Message
1 00:00.010 n/a Launching program /home/dschubert/code/examples/hello.
2 00:03.675 0-3 Startup complete.
▶ Stacks
3 00:08.067 0-3 Process stopped at breakpoint in main (hello.c:170).
▶ Stacks
▼ Stack for process 0
#0 main (argc=1, argv=0x7fffffffcf68, environ=0x7fffffffd290) at /home/dschubert/code/ddt/examples/hello.c:170
▼ Local variables for process 0 (ranges shown for 0-3)
Name Value
argc 1
argv 0x7fffffffcf68
beingWatched 3 (-1-3)
bigArray
dest 32767 (0-32767)
dynamicArray 0x80ba00
environ 0x7fffffffd290
i 1
message "Greetings from process 3!"
my_rank 0 (0-3)
p 4
s 0
source 4 (-153326624-4)
status
t2 0x603010
tables
tag 50
test
x 10000
y 12
4 00:08.299 n/a Every process in your program has terminated.
Messages
Tracepoints
Output
Tracepoints
Time Tracepoint Processes Values
1 00:04.512 hello.c:92 0-3 x: 0
2 00:04.513 hello.c:92 0-3 x: 1000
3 00:04.515 hello.c:92 0-3 x: 2000
4 00:05.409 hello.c:92 0-3 x: 3000
5 00:05.411 hello.c:92 0-3 x: 4000
6 00:05.413 hello.c:92 0-3 x: 5000
7 00:06.313 hello.c:92 0-3 x: 6000
8 00:06.315 hello.c:92 0-3 x: 7000
9 00:06.316 hello.c:92 0-3 x: 8000
10 00:07.215 hello.c:92 0-3 x: 9000
Messages
Tracepoints
Output
Output
(00:07.527) 2: my rank is 2
(00:07.528) 0: my rank is 0
(00:07.528) 3: my rank is 3
(00:07.529) 1: my rank is 1
(00:07.529) 0-3: sizeof(int) = 4
(00:07.529) 0-3: sizeof(void*) = 8
(00:07.530) 2: My pid is 16549.
(00:07.530) 0: My pid is 16548.
(00:07.531) 1: My pid is 16547.
(00:07.531) 3: My pid is 16550.
(00:07.532) 0-3: I have 1 arguments.
(00:07.532) 0-3: How many did I say?
(00:07.533) 0-3: They are:
(00:07.533) 0-3: 0: /home/dschubert/code/ddt/examples/hello
(00:07.572) 2-3: I can write to stderr too
(00:07.577) 2: sending message from (2)
(00:07.577) 3: sending message from (3)
(00:07.577) 1: sending message from (1)
(00:07.578) 0: waiting for message from (1)
(00:07.578) 0-1: I can write to stderr too
(00:07.578) 0: Greetings from process 1!
(00:07.578) 0: waiting for message from (2)
(00:07.578) 0: Greetings from process 2!
(00:07.579) 0: waiting for message from (3)
(00:07.579) 0: Greetings from process 3!
(00:09.020) 3: all done...(3)
(00:09.021) 2: all done...(2)
(00:09.022) 0: all done...(0)
(00:09.022) 1: all done...(1)

Figure 83: *Offline Mode HTML output*

Timestamps are recorded with the contents in the offline log, and even though the file is neatly organized into three sections, it remains possible to identify ordering of events from the time stamp.

The Messages section contains

- Error messages – for example if DDT's Memory Debugging detects an error then the message and the stack trace at the time of the error will be recorded from each offending processes. Where multiple processes error within a short interval, one messsage is generated with the parallel stack view including the affected processes.
- Recorded breakpoints – the stack traces, local variables and stack arguments of a pausing process. Where multiple processes pause within a short interval, then variables across the pausing processes will be compared and a spark line drawn, and the stacks displayed will be the merged parallel stack output from the pausing processes.

The *Tracepoints* section contains the output from tracepoints, similar to that shown in the tracepoints window in an online debugging session – including spark lines displaying the variable distribution.

© 2015 Allinea Software Ltd.

Output from the application will be written to the *Output* section. For most MPIs this will not be identifiable to a particular process, but on those MPIs that do support it, DDT will report which processes have generated the output.

Identical output from the *Output* and *Tracepoints* section will, if received in close proximity and order, be merged in the output, where this is possible.

15.3 Offline Report Output (Plain Text)

Unlike the offline report in HTML mode, the plain text mode does not separate the tracepoint, breakpoint and application output into separate sections.

Lines in the offline plain text report are identified as messages, standard output, error output, and tracepoints, as detailed in the Offline Report Output (HTML) section previously.

For example, a simple report could look like:

```
message (0-3): Process stopped at breakpoint in main (hello.c:97).
message (0-3): Stacks
message (0-3): Processes Function
message (0-3): 0-3    main (hello.c:97)
message (0-3): Stack for process 0
message (0-3): #0 main (argc=1, argv=0x7fffffffd378, \
        environ=0x7fffffffd388) at /home/ddt/examples/hello.c:97
message (0-3): Local variables for process 0 \
        (ranges shown for 0-3)
message (0-3): argc: 1 argv: 0x7fffffffd378 beingWatched: 0 \
        dest: 7 environ: 0x7fffffffd388 i: 0 message: ",!\312\t" \
        my_r ank: 0 (0-3) p: 4 source: 0 status: t2: 0x7ffff7ff7fc0 \
        tables: tag: 50 test: x: 10000 y: 12
```

© 2015 Allinea Software Ltd.

16 DDT: Using DDT with the VisIt Visualization Tool

VisIt (http://wci.llnl.gov/codes/visit/) provides large-scale distributed visualization of complex data structures, but typically requires program instrumentation to achieve online visualization. DDT may function as a VisIt data source, providing basic online visualization capabilities during a debugging session without the need for program instrumentation or recompilation.

16.1 Support for VisIt

Please note: *We can't provide support for building, setting up or general use of VisIt.* Please ensure that VisIt works with one of the example simulation clients before proceeding. VisIt support is available from:

- The visit-users@ornl.gov mailing list (http://elist.ornl.gov/mailman/listinfo/visit-users)
- The VisIt user community web site (http://visitusers.org)

16.2 Patching and Building VisIt

Versions of VisIt prior to 2.6 must be patched to work with DDT, VisIt versions 2.6 onwards must be built from source with the cmake `VISIT_DDT` option enabled. To use the VisIt Picks feature (see 16.8 Focusing on a Domain & VisIt Picks) you must be using VisIt >= 2.6, have applied the appropriate patch and have compiled VisIt with `VISIT_DDT` enabled.

VisIt patch tarballs can be found in the visit-patches subdirectory of your DDT installation. Extract the tarball into your VisIt source directory (e.g. `visit2.6.2/src`). If you have the program `quilt` installed then run `quilt` in that directory repeatedly until you get the message "`File series fully applied`". If `quilt` is not available you can apply the patches using `patch` with the following command:

```
ls patches/*.patch | xargs -n 1 cat | patch -p 0
```

16.3 Compatibility

Supported platforms: Linux only
Supported MPIs: Open MPI, MPICH 2 and Cray MPT.
Remote Launch is not supported.
Only dynamically linked executables are supported.

16.4 Enabling VisIt Support in DDT

VisIt support is configured on the *File* → *Options* → *VisIt* page of the DDT options window:

© 2015 Allinea Software Ltd.

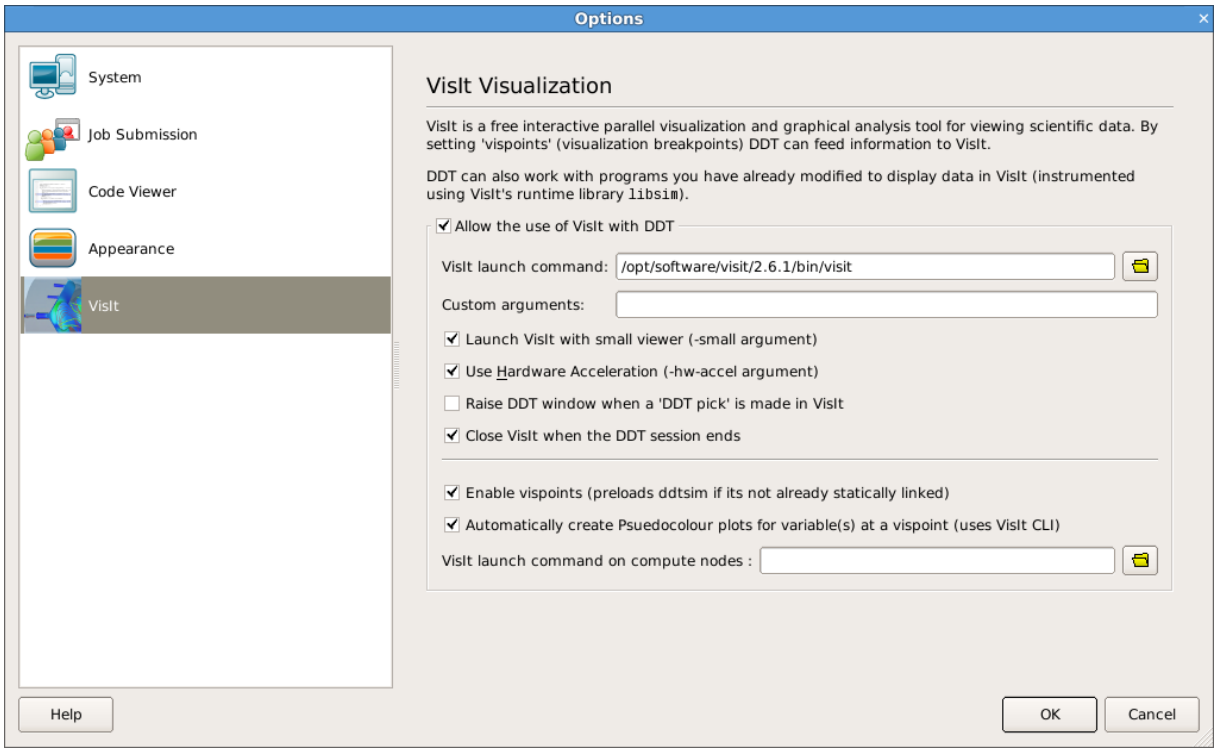

Options
System
Job Submission
Code Viewer
Appearance
VisIt
VisIt Visualization
VisIt is a free interactive parallel visualization and graphical analysis tool for viewing scientific data. By setting 'vispoints' (visualization breakpoints) DDT can feed information to VisIt.
DDT can also work with programs you have already modified to display data in VisIt (instrumented using VisIt's runtime library libsim).
Allow the use of VisIt with DDT
VisIt launch command: /opt/software/visit/2.6.1/bin/visit
Custom arguments:
Launch VisIt with small viewer (-small argument)
Use Hardware Acceleration (-hw-accel argument)
Raise DDT window when a 'DDT pick' is made in VisIt
Close VisIt when the DDT session ends
Enable vispoints (preloads ddtsim if its not already statically linked)
Automatically create Psuedocolour plots for variable(s) at a vispoint (uses VisIt CLI)
VisIt launch command on compute nodes :
Help
OK
Cancel

Figure 84: *VisIt Options Page*

1. Tick the *Allow the use of VisIt with DDT* box.
2. Enter the path to the `visit` command in the *VisIt Launch Command* box, e.g. `/opt/software/visit/2.6.1/bin/visit`.
3. Tick the *Enable visualization breakpoints (preloads ddtsim...)* option to enable DDT to work with programs that haven't been instrumented for use with VisIt.
4. Click *Ok*.

Hint: You can specify where the VisIt windows will be placed by adding geometry statements to the custom arguments line. The VisIt GUI is placed with `-geometry` argument, the viewer where your visualizations are drawn placed by `-viewer_geometry`. In both cases the geometry statements follow the form `[width]x[height]+[x-offset]+[y-offset]`. For example, the following positions the VisIt GUI and viewer to the right of the screen (depending on the size of your screen):

```
-geometry 150x800+1200+0 -viewer_geometry 500x500+700+250
```

Note: VisIt will apply the effects of the `-small` argument to after `-viewer_geometry` is applied, resulting in a viewer smaller than what you requested. We recommend you do not use the -small argument to VisIt if you are specifying window dimensions and locations manually with `-viewer_geometry`.

16.5 Setting Visualization Points (Vispoints)

A *vispoint* is a special breakpoint in DDT that, when hit, transfers control to VisIt to create a visual representation of an a 1-, 2- or 3-dimensional array.

To create a vispoint:

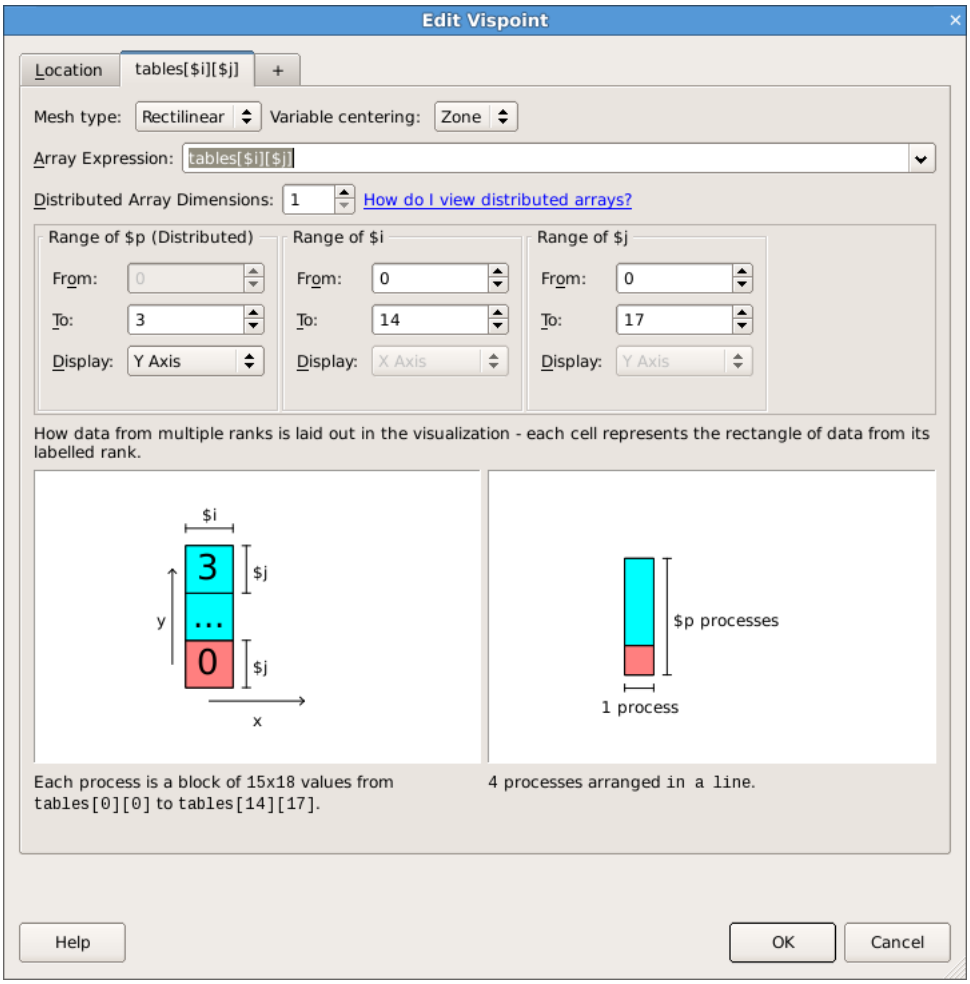


Figure 85: *Add/edit vispoint window*

1. Right-click on the line where you want to set it and select *Add Vispoint* from the menu.
2. If the mouse cursor is over an identifier representing an array, the vispoint may be pre-configured with it. If not, you have to enter the array variable (e.g. `tables`) manually followed by either (for C/C++) or ((for Fortran). DDT will try to auto-complete this expression (dimensions with bounds). If this is not possible one has to manually add `[$i][$j]`... (C/C++) or (`$i`, `$j`,...) (Fortran) to the expression and set the bounds. See section 7.15.1 Array Expression for more details.
3. Verify the arrays and processes will be arranged as you expect in VisIt by looking at the preview diagrams beneath the array expression configuration controls.
4. You can visualize additional arrays at this vispoint by clicking the + tab and entering another array expression.
5. Click *Ok*.

You can change the location and triggering conditions by switching to the *Vispoint* tab. Normal breakpoint constraints, such as *Trigger every n-th pass* may also be used in conjunction with Vispoints.

Note: Every time a vispoint is hit, it must be hit by all processes before either VisIt or DDT can continue. Position your vispoints carefully!

Note: DDT currently only supports one vispoint at a time. Changing a vispoint currently being displayed in VisIt is not recommended.

A vispoint can only visualize an array that is a continuous block in memory. C/C++ multi-dimensional

arrays on the heap (created with `malloc`, `calloc` or new) are arrays of pointers to arrays and cannot be visualized. If you have a one-dimensional C-style array (possibly allocated on the heap) you can visualise it in multiple dimensions using an array expression of the form:

```
myarray[$k*WIDTH*HEIGHT+$j*WIDTH+$i]
```

where `WIDTH` and `HEIGHT` are integers defining the dimensions of the 3D project to visualize (0 < $i <= `WIDTH` and 0 < $j >= `HEIGHT`). Note that when using a rectilinear mesh the ordering of the $i, $j, and $k terms in the array expression determines which dimension is mapped to the x, y and z axes in VisIt. For rectilinear meshes the dimension where adjacent cells are in adjacent memory locations ($i in the above array expression) *must* be mapped to the X-axis, and must therefore be the right-most term in a C-style code array expression.

16.6 Using Vispoints in DDT

Once you have added one or more vispoints to your program you can play it and when your program stops control will be transferred to VisIt.

1. Click *Play* to run your program as usual.
2. When a vispoint is hit the *Vispoint* window will appear:

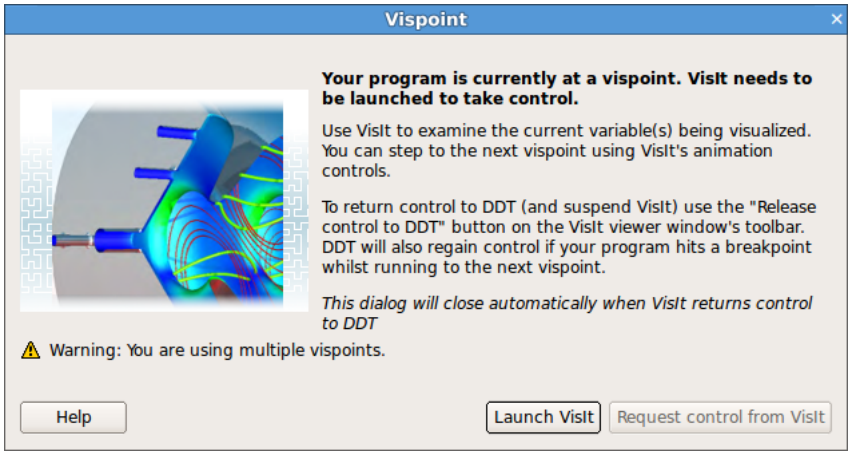

Vispoint
Your program is currently at a vispoint. VisIt needs to be launched to take control.
Use VisIt to examine the current variable(s) being visualized. You can step to the next vispoint using VisIt's animation controls.
To return control to DDT (and suspend VisIt) use the "Release control to DDT" button on the VisIt viewer window's toolbar. DDT will also regain control if your program hits a breakpoint whilst running to the next vispoint.
This dialog will close automatically when VisIt returns control to DDT
Warning: You are using multiple vispoints.
Help
Launch VisIt
Request control from VisIt

Figure 86: *Vispoint window*

1. Click the *Launch VisIt* button to start VisIt.
2. Wait for VisIt to load and connect to the simulation. The *Add* button in the *Plots* section of the VisIt GUI will light up when it's ready.
3. If *Automatically create Pseudocolour* plots is enabled (see 24.5.5) VisIt will automatically plot the arrays that were specified for the vispoint that was hit, each array in a different viewer window (whilst this is being done a VisIt CLI window will briefly appear).

   You can also manually add your plots in VisIt as normal. A *pseudocolor* plot is recommended, there will be a dataset with the same name as the array variable to be visualized. DDT makes available a mesh called `ddtmesh` which contains the data from the array selected in DDT, you can see how the data from each process is arranged on this mesh by using a *Subset* plot of *Domains*.

16.7 Returning to DDT

Either:

1. Click *Request control from* VisIt in the DDT *Vispoint* window; OR
2. Click the *Release to DDT* icon in the VisIt viewer window; OR
3. Stop the simulation and exit VisIt - the simulation will automatically be released back to DDT.

Note: DDT cannot regain control of simulation without an instruction from VisIt.

16.8 Focusing on a Domain & VisIt Picks

You may change the currently selected process in DDT by picking a zone or node directly from a VisIt plot.

Either:

1. Switch to the *DDT Pick* tool and click on a zone / node. The process that supplied the data for the selected zone / node will be selected in DDT; OR
2. Use the *Zone Pick* or *Node Pick* tool. From the pick window click *Focus In DDT*. This will switch to the process that supplied the data for the picked zone / node.

DDT will change its selected process and the status bar text will show which zone DDT was switched to. See section 16.7 Returning to DDT for how to get back to DDT and actually do something on this process.

Note: DDT will assume that the VisIt domain corresponds to the MPI rank of the process running that domain.

In VisIt 2.4.2, 2.5.0 and unpatched-2.6.x DDT will change its selected process and the status bar text will show which zone DDT was switched to; when using a patched version of VisIt 2.6.1 (see section 16.2 Patching and Building VisIt) data about the picked zone or node will be added to the VisIt Picks table – wait for this data to appear before making another pick in VisIt. See section 16.7 Returning to DDT for how to get back to DDT and actually do something on this process/data.

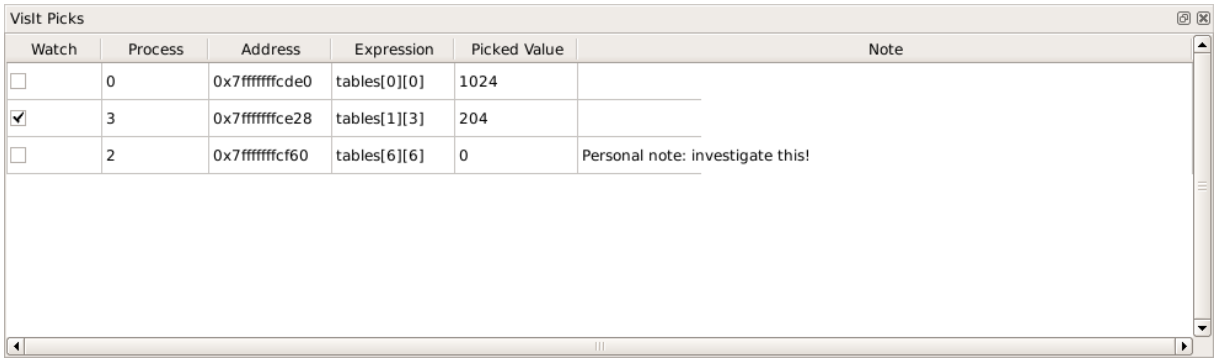


Figure 87: *VisIt Pick Window*

The *VisIt* Picks table displays information about the process and array index that supplied the data for the picked zone/node. This includes the process, the expression used to access the particular array element, the memory address of that array element and the value of that element as it was known to VisIt. You can set a personal reminder to identify a pick by double-clicking in the *Note* column. Clicking on a line in the table will focus DDT on the corresponding process, and you can create or remove a watchpoint on the relevant memory address by toggling the checkbox.

Note: There is a limit to the number of watchpoints that can be enabled at the same time.

© 2015 Allinea Software Ltd.

If you have enabled memory debugging (See section 11.1 Enabling Memory Debugging) you can get information concerning about the memory allocation the picked cell was a part of by right-clicking on an entry in the *VisIt Picks* table and selecting *View pointer details* (See section 11.4.2 View Pointer Details).

You can drag-drop a row of the VisIt Picks table into the *Evaluate Window* to see the current value for relevant array index (See section 7.4 Arbitrary Expressions And Global Variables). Remember the *Evaluate Window* will show the value for the currently selected process – the display will change as you switch focus to other processes.

16.9 Using DDT with a pre-instrumented program

Although DDT does not require your program to be instrumented for use with VisIt, DDT also supports existing programs that are already instrumented.

1. Debug your program with DDT using whatever arguments are required for your program to stop and connect to VisIt.
2. Launch VisIt, either as a separate instruction or with the *Launch VisIt* icon on the DDT toolbar.
3. Open your program's `.sim` file in VisIt from the normal place (by default `~/.visit/simulations`).
4. If launching VisIt from outside DDT, click *File* → *Connect to DDT* from within VisIt.

   Note: This is only necessary when launching VisIt yourself, when DDT launches VisIt it automatically connects.
5. Focus on a domain at any time (See section 16.8 Focusing on a Domain & VisIt Picks). Note that VisIt will be frozen when your simulation is paused in DDT. There will be no *Vispoint window* - just use VisIt whilst your simulation is running.

Note: DDT will assume that the VisIt domain corresponds to the MPI rank of the process running that domain.

© 2015 Allinea Software Ltd.

17 MAP: Starting

When compiling the program that you wish to profile, you must add the debug flag to your compile command. For the most compilers this is `-g`. You can use all optimisations that are compatible with the `-g` option. If your program is already compiled without debug information you will need to make the files that you are interested in again.

To start MAP simply type one of the following into a shell window:

```
map
map program_name
map program_name arguments
```

Note: You should not attempt to pipe input directly to MAP – for information about how to achieve the effect of sending input to your program, please read section 8 DDT: Program Input And Output.

Once started it will display the Welcome Page:

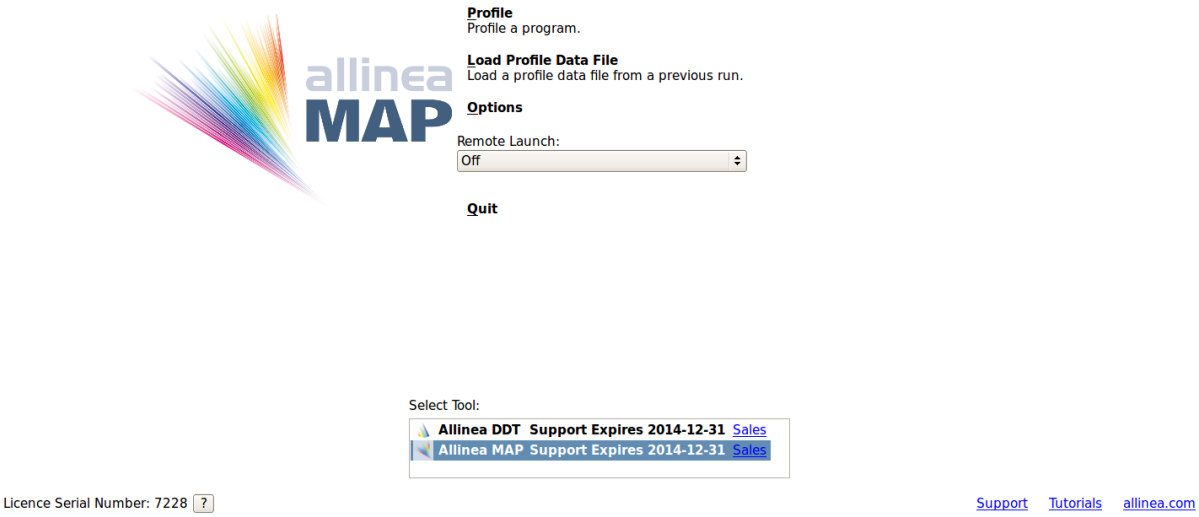


Figure 88: *MAP Welcome Page*

The *Welcome Page* allows you to choose what kind of profiling you want to do. You can:

- Profile a program
- Load a Profile from a previous run

17.1 Preparing a Program for Profiling

In most cases if your program is already compiled with debugging symbols then you do not need to recompile your program to use it with MAP although in some cases it may need to be relinked – this is explained in section 17.1.3 Linking below.

17.1.1 Debugging Symbols

Your program must be compiled with debugging symbols. For most compilers passing the `-g` option to the compiler is sufficient, e.g.

```
mpicc hello.c -o hello -g
```

Cray Compiler

For the Cray compiler we recommend using the -G2 option with MAP.

17.1.2 .eh-frame-hdr section

For statically-linked programs you may need to compile with extra flags to ensure the executable still has all the information MAP needs to record the call path while profiling it and gather the data needed for the Parallel Stack View. For the GNU linker this means adding `--Wl,--eh-frame-hdr` to the compile line or just `--eh-frame-hdr` to the link line. e.g.:

```
mpicc hello.c -o hello -g -Wl,--eh-frame-hdr
```

17.1.3 Linking

To collect data from your program, MAP provides two small libraries - `map-sampler` and `map-sampler-pmpi`. These must be linked with your program. On most systems MAP can do this automatically without any action by you. This is done via the system's `LD_PRELOAD` mechanism, which allows us to place an extra library into your program when starting it.

This automatic linking when starting your program only works if your program is dynamically-linked. Programs may be dynamically-linked or statically-linked, and for MPI programs this is normally determined by your MPI library. Most MPI libraries are configured with `--enable-dynamic` by default, and `mpicc/mpif90` produce dynamically-linked executables that MAP can automatically collect data from.

If MAP warns you that you have a statically-linked MPI executable, this often means your MPI library was not configured with `--enable-dynamic`. You now have three options:

1. Try compiling and linking your code dynamically. On most platforms this allows MAP to use the `LD_PRELOAD` mechanism to automatically insert its libraries into your application at runtime. This is not currently supported on Cray systems; you will need to use one of the following two options instead.
2. Link MAP's `map-sampler` and `map-sampler-pmpi` libraries with your program at link time manually. See 17.1.4 Static Linking and 17.1.5 Static Linking on Cray X-Series Systems.
3. Finally, it may be that your system supports dynamic linking but you have a statically-linked MPI. You can try to recompile the MPI implementation with `--enable-dynamic`, or find a dynamically version on your system, then recompile your program using that version. This will produce a dynamically-linked program that MAP can automatically collect data from.

17.1.4 Static Linking

If you compile your program statically (i.e. your MPI uses a static library or you pass the `-static` option to the compiler) then you must explicitly link your program with the MAP sampler and MPI wrapper libraries.

© 2015 Allinea Software Ltd.

Compiling the MAP MPI Wrapper Library

First you must compile the MAP MPI wrapper library for your system using the `make-map-static-libraries` command:

```
user@login:~/myprogram$ make-map-static-libraries

Created the MAP libraries in /users/ddt/allinea
   libmap-sampler.a
   libmap-sampler-pmpi.a

To instrument a program, add these compiler options:
   compilation : -g (and -O3 etc.)
       linking : -L/users/ddt/allinea -lmap-sampler-pmpi \
                 -Wl,--undefined,allinea_init_sampler_now \
                 -lmap-sampler -lstdc++ -lgcc_eh \
                 -Wl,--whole-archive -lpthread
                 -Wl,--no-whole-archive \
                 -Wl,--eh-frame-hdr
```

Linking with the MAP MPI Wrapper Library

Then when linking your program you must add the path to the MAP libraries (`-L/path/to/map-libraries`), and the libraries themselves (`-lmap-sampler-pmpi`, `-lmap-sampler`).

The MAP MPI wrapper library (`-lmap-sampler-pmpi`) must go:

1. *After* your program's object (`.o`) files.
2. *After* your program's own static libraries (e.g. `-lmylibrary`).
3. *After* the path to the MAP libraries (`-L/path/to/map-libraries`).
4. *Before* the MPI's Fortran wrapper library (e.g. `-lmpichf`) (if any).
5. *Before* the MPI's implementation library (usually `-lmpi`).
6. *Before* the MAP sampler library (`-lmap-sampler`).

The MAP sampler library (`-lmap-sampler`) must go:

1. *After* the MAP MPI wrapper library.
2. *After* your program's object (`.o`) files.
3. *After* your program's own static libraries (e.g. `-lmylibrary`).
4. *After* `-Wl,--undefined`,allinea_init_sampler_now.
5. *After* the path to the MAP libraries (`-L/path/to/map-libraries`).
6. *Before* `-lstdc++`, `-lgcc_eh`, `-lrt`, `-lpthread` and `-lc`.

Example:

```
mpicc hello.c -o hello -g -L/users/ddt/allinea \
   -lmap-sampler-pmpi \
   -Wl,--undefined,allinea_init_sampler_now \
   -lmap-sampler -lstdc++ -lgcc_eh \
   -Wl,--whole-archive -lpthread \
   -Wl,--no-whole-archive \
```

```
   -Wl,--eh-frame-hdr

mpif90 hello.f90 -o hello -g -L/users/ddt/allinea \
   -lmap-sampler-pmpi \
   -Wl,--undefined,allinea_init_sampler_now \
   -lmap-sampler -lstdc++ -lgcc_eh
   -Wl,--whole-archive -lpthread \
   -Wl,--no-whole-archive \
   -Wl,--eh-frame-hdr
```

MVAPICH 1

You must add `-lmpichf` after `-lmap-sampler-pmpi` (MVAPICH must be compiled with Fortran support).

If you get a linker error about multiple definitions of `mpi_init_`, you need to specify additional linker flags:

```
-Wl,--allow-multiple-definition
```

17.1.5 Static Linking on Cray X-Series Systems

Compiling the MAP MPI Wrapper Library

On Cray X-Series systems use the `make-map-static-cray-libraries` instead:

```
Created the MAP libraries in /users/ddt/allinea:
   libmap-sampler.a
   libmap-sampler-pmpi.a

To instrument a program, add these compiler options:
   compilation : -g (or '-G2' for native Cray Fortran)
                 (and -O3 etc.)
       linking : -L/users/ddt/allinea -lmap-sampler-pmpi \
                 -Wl,--undefined,allinea_init_sampler_now \
                 -lmap-sampler -lstdc++ -lgcc_eh \
                 -Wl,--whole-archive -lpthread \
                 -Wl,--no-whole-archive \
                 -Wl,--eh-frame-hdr
```

Linking with the MAP MPI Wrapper Library

```
cc hello.c -o hello -g -L/users/ddt/allinea \
   -lmap-sampler-pmpi \
   -Wl,--undefined,allinea_init_sampler_now \
   -lmap-sampler -lstdc++ -lgcc_eh \
   -Wl,--eh-frame-hdr

ftn hello.f90 -o hello -g -L/users/ddt/allinea \
   -lmap-sampler-pmpi \
   -Wl,--undefined,allinea_init_sampler_now \
   -lmap-sampler -lstdc++ -lgcc_eh \
   -Wl,--eh-frame-hdr
```

© 2015 Allinea Software Ltd.

17.1.6 Manual Dynamic Linking on Cray X-Series systems

Compiling the MAP MPI Wrapper Library

On Cray X-Series systems use the `make-map-shared-cray-libraries` command:

```
Created the MAP libraries in /users/ddt/allinea:
   libmap-sampler.so       (and .so.1, .so.1.0, .so.1.0.0)
   libmap-sampler-pmpi.so  (and .so.1, .so.1.0, .so.1.0.0)

To instrument a program, add these compiler options:
   compilation : -g (or '-G2' for native Cray Fortran) (and -O3
      etc.)
       linking : -dynamic -L/users/ddt/allinea lmap-sampler-pmpi -
          lmap-sampler -Wl,--eh-frame-hdr

Note: These libraries must be on an NFS/Lustre/GPFS partition,
   just like your program.

Before running aprun (interactively or from a queue), set LD\
   _LIBRARY\_PATH:
   export LD_LIBRARY_PATH=/users/ddt/allinea:$LD_LIBRARY_PATH
   aprun  ...
```

Linking with the MAP MPI Wrapper Library

```
cc hello.c -o hello -g -dynamic -L/users/ddt/allinea \
   -lmap-sampler-pmpi -lmap-sampler \
   -Wl,--eh-frame-hdr

ftn hello.f90 -o hello -dynamic -g -L/users/ddt/allinea \
   -lmap-sampler-pmpi -lmap-sampler \
   -Wl,--eh-frame-hdr
```

Unlike static linking, the order in which libraries are specified is not important. Remember to set `LD_LIBRARY_PATH` to include the directory containing the Allinea sampler libraries before running your application:

```
export LD_LIBRARY_PATH=/users/ddt/allinea:\$LD_LIBRARY_PATH
```

© 2015 Allinea Software Ltd.

17.2 Profiling a Program

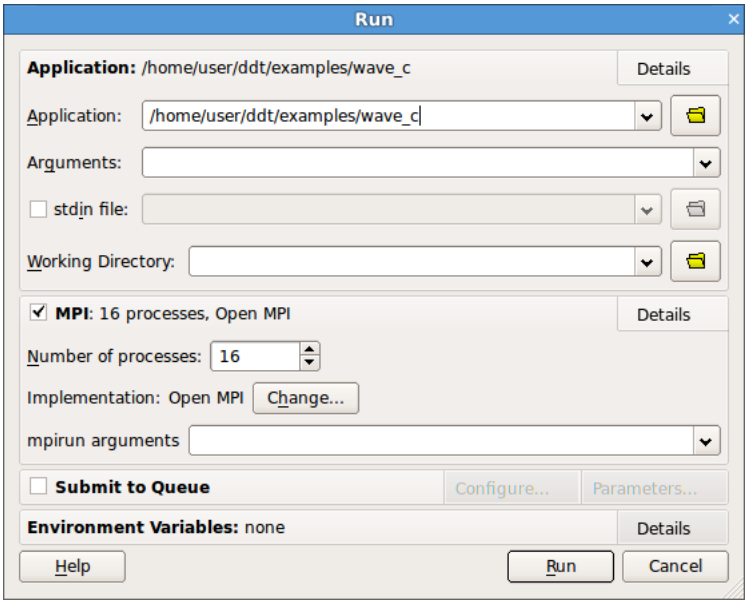


Figure 89: *Run window*

If you click the *Profile* button on the MAP *Welcome Page* you will see the window above. The settings are grouped into sections. Click the *Details...* button to expand a section. The settings in each section are described below.

17.2.1 Application

Application: The full path name to your application. If you specified one on the command line, this will already be filled in. You may browse for an application by clicking on the *Browse* button.

Note: Many MPIs have problems working with directory and program names containing spaces. We recommend avoiding the use of spaces in directory and file names.

Arguments: (optional) The arguments passed to your application. These will be automatically filled if you entered some on the command line.

Note: Avoid using quote characters such as ' and “, as these may be interpreted differently by MAP and your command shell. If you must use these and cannot get them to work as expected, please contact support@allinea.com .

stdin file: (optional) This allows you to choose a file to be used as the standard input (`stdin`) for your program. MAP will automatically add arguments to `mpirun`to ensure your input file is used.

Working Directory: (optional) The working (i.e. current directory) to use when running your application. If this is blank then MAP's working directory will be used instead.

17.2.2 MPI

Note: If you only have a single process licence or have selected none as your MPI Implementation *the* MPI *options will be missing. The* MPI *options are not available when in single process mode. See section 17.4 Profiling Single-Process Program for more details about using a single process.*

Number of processes: The number of processes that you wish to profile. MAP supports hundreds of thousands of processes but this is limited by your licence. This option may not be displayed if disabled on the *Job Submission* options page.

Number of nodes: This is the number of compute nodes that you wish to use to run your program. This option is only displayed for certain MPI implementations or if it is enabled on the *Job Submission* options page.

Processes per node: This is the number of MPI processes to run on each compute node. This option is only displayed for certain MPI implementations or if it is enabled on the *Job Submission* options page.

Implementation: The MPI implementation to use, e.g. Open MPI, MPICH 2 etc. Normally the Auto setting will detect the currently loaded MPI module correctly. If you are submitting a job to a queue the queue settings will also be summarised here. You may change the MPI implementation by clicking on the *Change...* button.

Note: The choice of MPI implementation is critical to correctly starting MAP. Your system will normally use one particular MPI implementation. If you are unsure as to which to pick, try generic, consult your system administrator or Allinea. A list of settings for common implementations is provided in B MPI Distribution Notes and Known Issues.

Note: If your desired MPI command is not in your `PATH`*, or you wish to use an MPI run command that is not your default one, you can configure this using the Options window (See section 24.5.1 System).*

mpirun arguments: (optional): The arguments that are passed to `mpirun`or your equivalent – usually prior to your executable name in normal `mpirun`usage. You can place machine file arguments – if necessary – here. For most users this box can be left empty.

Note: You should ***not*** *enter the* `-np` *argument as MAP will do this for you.*

17.2.3 Environment Variables

The optional Environment Variables section should contain additional environment variables that should be passed to `mpirun`or its equivalent. These environment variables may also be passed to your program, depending on which MPI implementation your system uses. Most users will not need to use this box.

17.2.4 Profiling

Click *Run* to start your program – or *Submit* if working through a queue (See section 24.2 Integration With Queuing Systems). This will compile up a MPI wrapper library on the fly that can intercept the `MPI_INIT` call and gather statistics about MPI use in your program. If this has problems see E.9.2 MPI Wrapper Libraries. Then MAP brings up the Running window and starts to connect to your processes.

The program runs inside MAP which starts collecting stats on your program through the MPI interface you selected and will allow your MPI implementation to determine which nodes to start which processes on.

MAP collects data for the entire program run by default – our sampling algorithms ensure only a few tens of megabytes are collected even for very long-running jobs. If you wish, you can stop your program at any time by using the *Stop and Analyze* button – MAP will then collect the data recorded so far, stop your program and end the MPI session before showing you the results. If any processes remain you

© 2015 Allinea Software Ltd.

may have to clean them up manually using the `kill` command (or a command provided with your MPI implementation), but this should not be necessary.

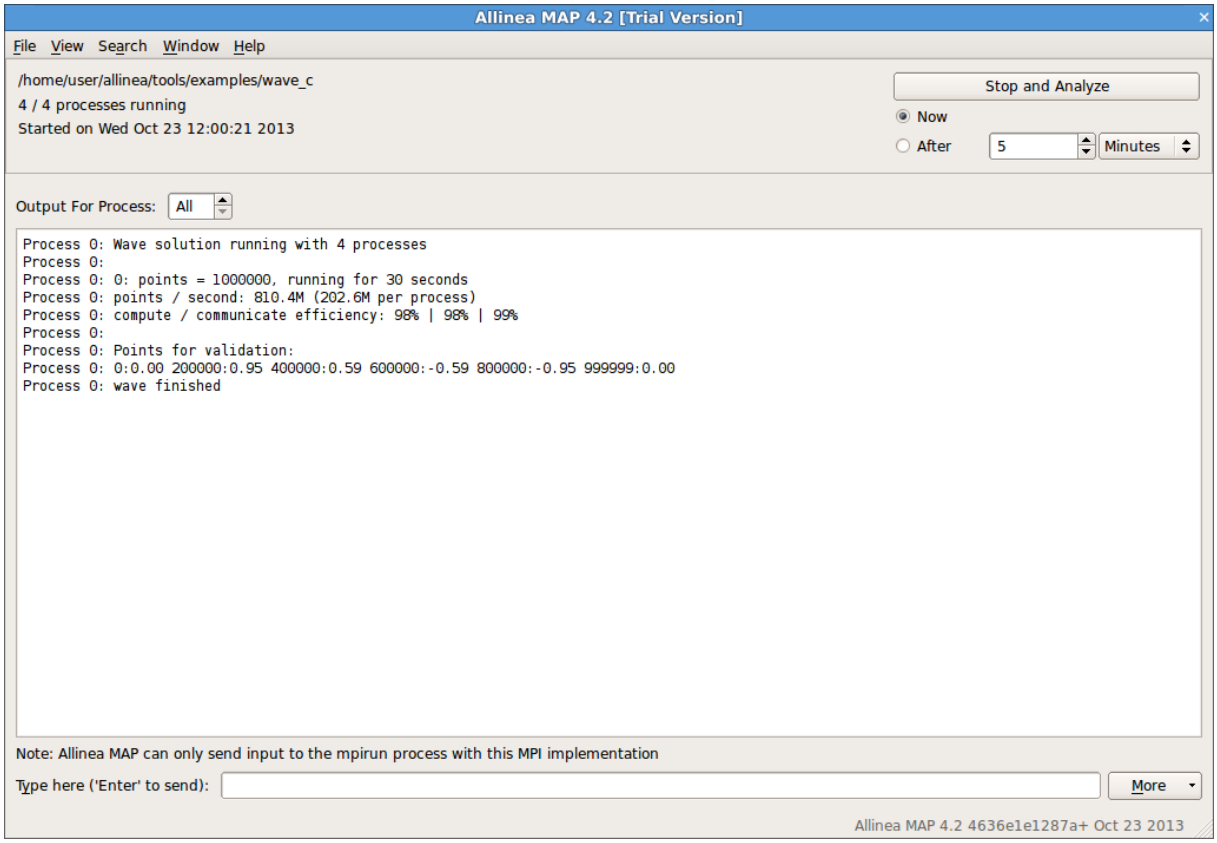

Allinea MAP 4.2 [Trial Version]
File View Search Window Help
/home/user/allinea/tools/examples/wave_c
4 / 4 processes running
Started on Wed Oct 23 12:00:21 2013
Stop and Analyze
Now
After
5
Minutes
Output For Process:
All
Process 0: Wave solution running with 4 processes
Process 0:
Process 0: 0: points = 1000000, running for 30 seconds
Process 0: points / second: 810.4M (202.6M per process)
Process 0: compute / communicate efficiency: 98% | 98% | 99%
Process 0:
Process 0: Points for validation:
Process 0: 0:0.00 200000:0.95 400000:0.59 600000:-0.59 800000:-0.95 999999:0.00
Process 0: wave finished
Note: Allinea MAP can only send input to the mpirun process with this MPI implementation
Type here ('Enter' to send):
More
Allinea MAP 4.2 4636e1e1287a+ Oct 23 2013

Figure 90: *Running window*

17.3 `remote-exec` Required By Some MPIs

When using Open MPI, SGI MPT, MPICH 1 Standard or the MPMD variants of MPICH 2, MPICH 3 or Intel MPI, MAP will allow `mpirun`to start all the processes, then attach to them while they're inside `MPI_Init`.

This method is often faster than the generic method, but requires the `remote-exec` facility in MAP to be correctly configured if processes are being launched on a remote machine. For more information on remote-exec, please see section 24.4 Connecting to remote programs (remote-exec).

Important: If MAP is running in the background (e.g. `map` &) then this process may get stuck (some SSH versions cause this behaviour when asking for a password). If this happens to you, go to the terminal and use the fg or similar command to make MAP a foreground process, or run MAP again, without using "&".

If MAP can't find a password-free way to access the cluster nodes then you will not be able to use the specialised startup options. Instead, You can use *generic*, although startup may be slower for large numbers of processes.

© 2015 Allinea Software Ltd.

17.4 Profiling Single-Process Program

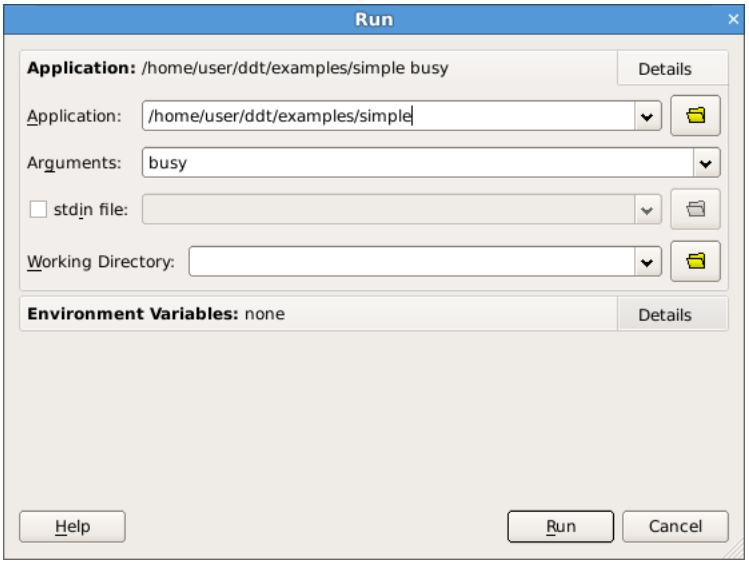


Figure 91: *Single-Process Run Window*

Users with single-process licences will immediately see the *Run Window* that is appropriate for single-process applications.

Users with multi-process licences can check the *Run Without MPI Support* checkbox to run a single process program.

Select the application – either by typing the file name in, or selecting using the browser by clicking the browse button. Arguments can be typed into the supplied box.

Finally click *Run* to start your program.

17.5 Sending Standard Input

MAP provides a *stdin file* box in the *Run* window. This allows you to choose a file to be used as the standard input (stdin) for your program. (MAP will automatically add arguments to mpirun to ensure your input file is used.)

Alternatively, you may enter the arguments directly in the *mpirun Arguments* box. For example, if using MPI directly from the *command-line* you would normally use an option to the mpirun such as `-stdin filename`, then you may add the same options to the *mpirun Arguments* box when starting your MAP session in the *Run* window.

It is also possible to enter input during a session. Start your program as normal, then switch to the *Input/Output* panel. Here you can see the output from your program and type input you wish to send. You may also use the *More* button to send input from a file, or send an EOF character.

© 2015 Allinea Software Ltd.

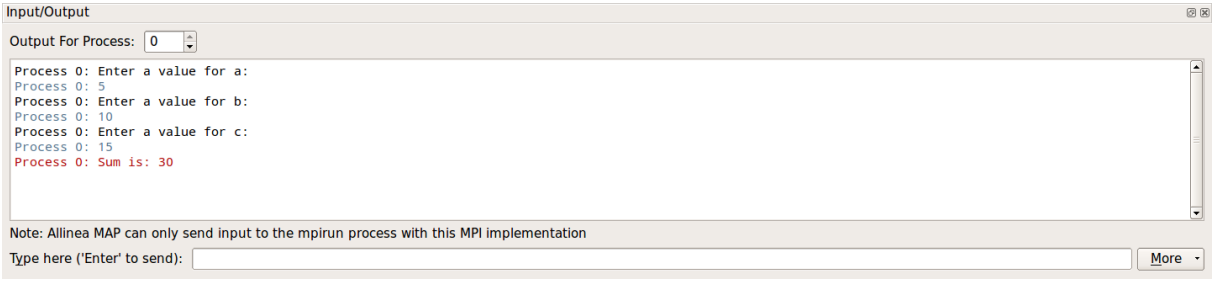

Input/Output
Output For Process: 0
Process 0: Enter a value for a:
Process 0: 5
Process 0: Enter a value for b:
Process 0: 10
Process 0: Enter a value for c:
Process 0: 15
Process 0: Sum is: 30
Note: Allinea MAP can only send input to the mpirun process with this MPI implementation
Type here ('Enter' to send):
More

Figure 92: *MAP Sending Input*

Note: If MAP is running on a fork-based system such as Scyld, or a `-comm=shared` *compiled MPICH 1, your program may not receive an EOF correctly from the input file. If your program seems to hang while waiting for the last line or byte of input, this is likely to be the problem. See E General Troubleshooting and Known Issues or contact Allinea for a list of possible fixes.*

17.6 Starting A Job In A Queue

If MAP has been configured to be integrated with a queue/batch environment, as described in section 24.2 Integration With Queuing Systems then you may use MAP to launch your job. In this case, a Submit button is presented on the Run Window, instead of the ordinary Run button. Clicking Submit from the Run Window will display the queue status until your job starts. MAP will execute the display command every second and show you the standard output. If your queue display is graphical or interactive then you cannot use it here.

If your job does not start or you decide not to run it, click on Cancel Job . If the regular expression you entered for getting the job id is invalid or if an error is reported then MAP will not be able to remove your job from the queue – it is strongly recommend you check the job has been removed before submitting another as it is possible for a forgotten job to execute on the cluster and either waste resources or interfere with other profiling sessions.

After the sampling (program run) phase is complete, MAP will start the analysis phase, collecting and processing the distinct samples. This could be a lengthy process depending on the size of the program. For very large programs it could be as much as 10 or 20 minutes.

You should ensure that your job does not hit its queue limits during the analysis process, setting the job time large enough to cover both the sampling and the analysis phases.

MAP will also require a little extra memory both in the sampling and in the analysis phases. Please, ensure the job memory allocation is large enough to handle this.

Once your job is running, it will connect to MAP and you will be able to profile it.

17.7 Using Custom MPI Scripts

On some systems a custom '`mpirun`' replacement is used to start jobs, such as `mpiexec`. MAP will normally use whatever the default for your MPI implementation is, so for MPICH 1 it would look for `mpirun`and not `mpiexec`, for SLURM it would use `srun` etc. This section explains how to configure MAP to use a custom `mpirun`command for job start up.

There are typically two ways you might want to start jobs using a custom script, and MAP supports them both. Firstly, you might pass all the arguments on the command-line, like this:

```
mpiexec -n 4 /home/mark/program/chains.exe /tmp/mydata
```

There are several key variables in this line that MAP can fill in for you:

1. The number of processes (4 in the above example)
2. The name of your program (`/home/mark/program/chains.exe`)
3. One or more arguments passed to your program (`/tmp/mydata`)

Everything else, like the name of the command and the format of it's own arguments remains constant. To use a command like this in MAP, we adapt the queue submission system described in the previous section. For this `mpiexec`example, the settings would be as shown here:

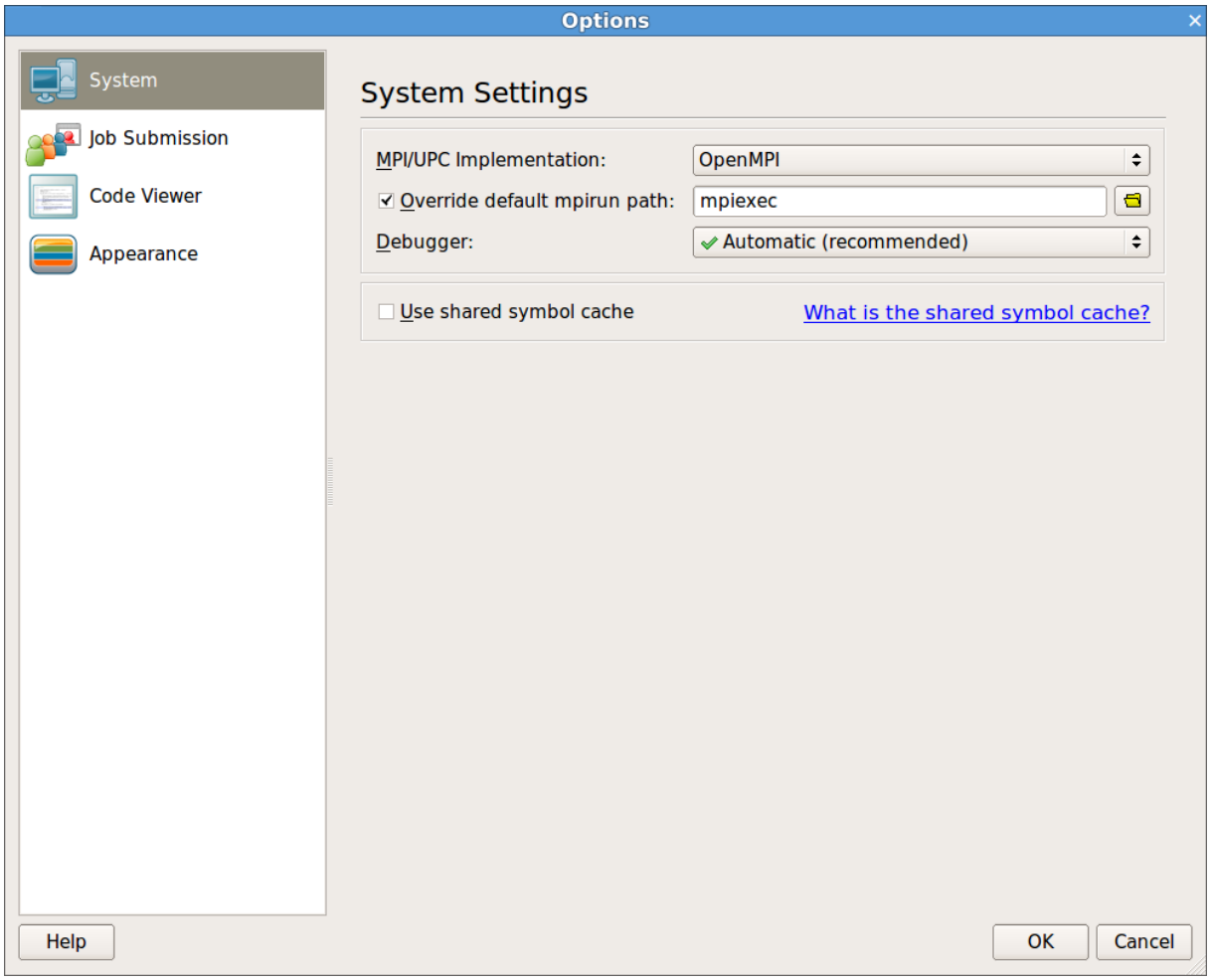


Figure 93: *MAP Using Custom MPI Scripts*

As you can see, most of the settings are left blank. Let's look at the differences between the *Submit Command* in MAP and what you would type at the command-line:

1. The number of processes is replaced with `NUM_PROCS_TAG`
2. The name of the program is replaced by the full path to `ddt-debugger` (used by both DDT and MAP)
3. The program arguments are replaced by `PROGRAM_ARGUMENTS_TAG`

Note, it is NOT necessary to specify the program name here. MAP takes care of that during its own startup process. The important thing is to make sure your MPI implementation starts `ddt-debugger` instead of your program, but with the same options.

© 2015 Allinea Software Ltd.

The second way you might start a job using a custom `mpirun`replacement is with a settings file:

```
mpiexec -config /home/mark/myapp.nodespec
```

where `myfile.nodespec` might contains something like this:

```
comp00 comp01 comp02 comp03 : /home/mark/program/chains.exe /tmp/
   mydata
```

MAP can automatically generate simple configuration files like this every time you run your program – you just need to specify a template file. For the above example, the template file `myfile.template` would contain the following:

```
comp00 comp01 comp02 comp03 : DDTPATH_TAG/bin/ddt-debugger
   DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_ARGUMENTS_TAG
```

This follows the same replacement rules described above and in detail in section 24.2 Integration With Queuing Systems. The options settings for this example might be:

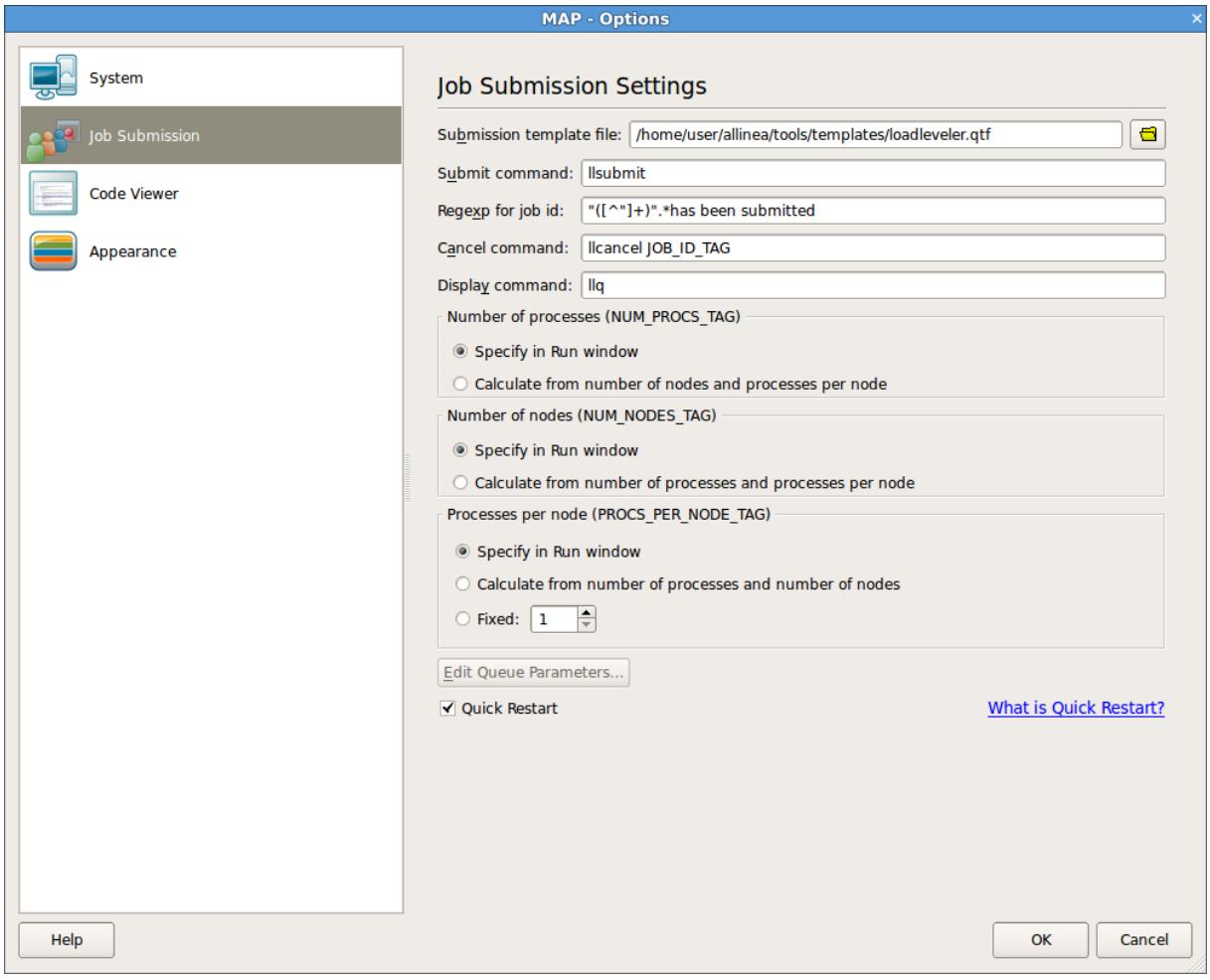


Figure 94: *MAP Using Substitute MPI Commands*

Note the Submit Command and the Submission Template File in particular. MAP will create a new file and append it to the submit command before executing it. So, in this case what would actually be executed might be `mpiexec -config /tmp/allinea-temp-0112` or similar. Therefore, any argument like `-config` must be last on the line, because MAP will add a file name to the end of the line. Other arguments, if there are any, can come first.

© 2015 Allinea Software Ltd.

We recommend reading the section on queue submission, as there are many features described there that might be useful to you if your system uses a non-standard start up command. If you do use a non-standard command, please email us at support@allinea.com and let us know about it – you might find the next version supports it out-of-the-box!

17.8 Starting MAP From A Job Script

While its common when debugging to submit runs from inside a debugger, for profiling the usual approach would be to run the program offline, producing a profile file that can be inspected later. To do this replace your usual program invocation with a MAP command. So

```
mpirun -n 4 PROGRAM [ARGUMENTS]...
```

becomes

```
map -profile -n 4 PROGRAM [ARGUMENTS]...
```

MAP will run without a GUI, gathering data to a `.map` profile file. Its filename is based on a combination of program name, processor count and timestamp, like `program_2p_2012-12-19_10-51.map` although this may be changed with the `-output` argument. To examine this file, either run MAP and select the *Load Profile Data File* option, or access it directly with command

```
map program_2p_2012-12-19_10-51.map
```

By default, when running without a GUI, MAP will print some messages and prefix each line of your program's output with the rank. The `-silent` argument suppresses this additional output so your program's output is intact.

17.9 MAP Environment Variables

MAP_INTERVAL

MAP takes a sample in each 20ms period, giving it a default sampling rate of 50Hz. This will be automatically decreased as the run proceeds to ensure a constant number of samples are taken (see `MAP_NUM_SAMPLES`). If your program runs for a very short period of time, you may benefit by decreasing the initial sampling interval. For example, `MAP_INTERVAL=1` sets an initial sampling rate of 1000Hz, or once per millisecond. Higher sampling rates are not supported.

MAP_KEEP_SAMPLES_FILES

MAP samples are temporarily written to file. By default MAP deletes the sample file once it has been re-read. To keep the sample file set `MAP_KEEP_SAMPLES_FILES=<directory>` and MAP will move the sample file into the given directory. The file will be suffixed with .keep.

MAP_MPI_WRAPPER

To direct MAP to use a pre-compiled wrapper instead of generating one on-the-fly set `MAP_MPI_WRAPPER=<pathofsharedobject>`. To generate the wrapper set `MPICC` and run `<path to MAP installation>/map/wrapper/build\_wrapper` which will generate the wrapper library `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so` with symlinks to `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1`, `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1.0`, and `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1.0.0`.

MAP_MPIRUN

The path of `mpirun`, `mpiexec`or equivalent. If this is set it has higher priority than that set in the GUI and the `mpirun`found in PATH.

MAP_NUM_SAMPLES

MAP collects 1000 samples per process by default. To avoid generating too much data on long runs, the sampling rate will be automatically decreased as the run progresses to ensure only 1000 evenly-spaced samples are stored. You may adjust this by setting `MAP_NUM_SAMPLES=<positiveinteger>` - however, we strongly recommend leaving this value at the default setting - higher values are not generally beneficial and add extra memory overheads while running your code. Bear in mind that with 512 processes, the default setting already collects half a million samples over the job - the effective sampling rate can be very high indeed.

MAP_PRESERVE_WRAPPER

To gather data from MPI calls MAP generates a wrapper to the chosen MPI implementation (see 17.1 Preparing a Program for Profiling). By default the generated code and shared objects are deleted when MAP no longer needs them. To prevent MAP from deleting these files set `MAP_PRESERVE_WRAPPER=1`. Please note that if you are using remote launch then this variable must be exported in the remote script (see Error: Reference source not found Error: Reference source not found).

MAP_SAMPLER_NO_TIME_MPI_CALLS

Set this to prevent MAP from timing the time spent in MPI calls.

MAP_SAMPLER_TRY_USE_SMAPS

Set this to allow MAP to use `/proc/[pid]/smaps` to gather memory usage data. This is not recommended since it slows down sampling significantly.

MPICC

To create the MPI wrapper MAP will try to use `MPICC`, then if that fails search for a suitable MPI compiler command in `PATH`. If the MPI compiler used to compile the target binary is not in `PATH` (or if there are multiple MPI compilers in `PATH`) then `MPICC` should be set.

© 2015 Allinea Software Ltd.

18 MAP: Program Output

MAP collects and displays output from all processes under the *Input/Output* tab. Both standard output and error are shown. As the output is shown after the program has completed, there are not the problems with buffering that occur with DDT.

18.1 Viewing Standard Output And Error

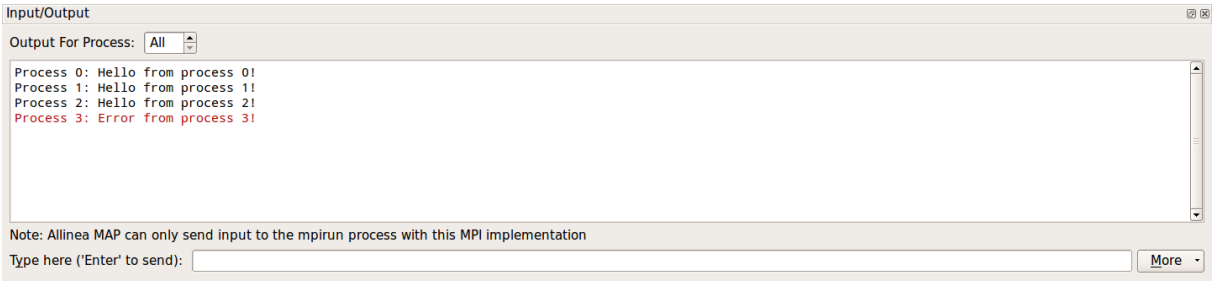

Input/Output
Output For Process:
All
Process 0: Hello from process 0!
Process 1: Hello from process 1!
Process 2: Hello from process 2!
Process 3: Error from process 3!
Note: Allinea MAP can only send input to the mpirun process with this MPI implementation
Type here ('Enter' to send):
More

Figure 95: *MAP Standard Output Window*

The Input/Output tab is at the bottom of the screen (by default).

The output may be selected and copied to the X-clipboard.

18.2 Displaying Selected Processes

You can choose whether to view the output for all processes, or just a single process.

Note: Some MPI implementations pipe stdin, stdout and stderr from every process through mpirun or rank 0.

18.3 Saving Output

By right-clicking on the text it is possible to save it to a file. You also have the option to copy a selection to the clipboard.

© 2015 Allinea Software Ltd.

19 MAP: Source Code View

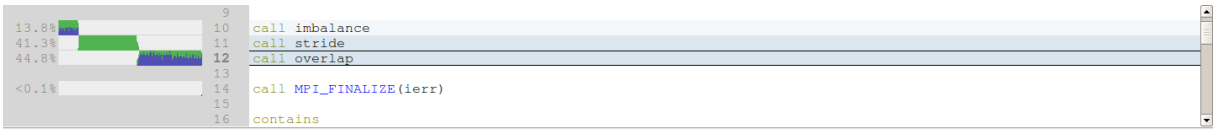


Figure 96: *Source Code View*

The centre pane shows your source code, annotated with performance information:

Beside each line of code is a time chart, showing how much total time was spent computing (green) and communicating (blue) on that line. Above we see that this call to overlap took 20.4% of the total runtime. The vertical axis is the number of processes and the horizontal is wall clock time. `stride` is pure compute, `late` pure comms, `overlap` does both.

Only 'interesting' lines get charts - lines in which at least 0.1% of the program's total time was spent. Finding these by scrolling around can be burdensome - for that you can use the *Stacks* view.

© 2015 Allinea Software Ltd.

20 MAP: Parallel Stack View

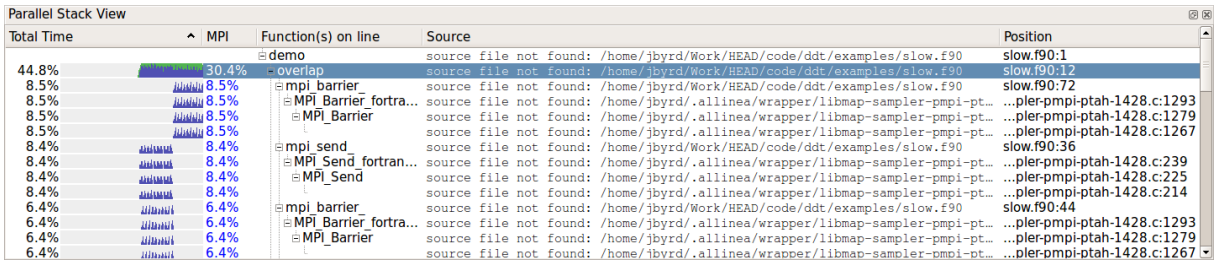


Figure 97: *MAP Parallel Stack View*

The *Stacks* view is a more classic profiler view, with a twist. Most profilers would show you a list or graph of functions; in MAP each line in the tree refers to a specific line of source code.

So, where you see `MPI_Barrier` in here it doesn't represent all combined `MPI_Barrier` calls, but rather the wall clock time from one specific line in your program - which is shown in the *Source* column. Clicking on any line also jumps the code view to that position, too.

The percentage MPI time will give you an idea as to how well your program is scaling, and the *Stacks* view will show which lines of code spend the most time running, computing or waiting - as with most places in the GUI you can hover over a line or chart for more a more detailed breakdown.

The *Stacks* view offers a good top-down view of your program; it's easy to follow down from the main function to see which code paths took the most time.

© 2015 Allinea Software Ltd.

21 MAP: Project Files View

Figure 98: *Project files view*

The *Project Files* view helps you in two ways: firstly it's a great way to browse around and navigate through a large, unfamiliar code base. Secondly, it offers a bottom-up view of the performance of your program. Each file, function or folder comes with a time chart that shows how much wall clock time was spent executing code inside that file / function / folder:

Files with multiple routines, like `slow.f90` can be expanded. Note it shows the time spent in the routine (and external routines it calls), but not in other application code routines, so the top level program slow is only at 0.0% The *Project Files* view helps you find specific folders, files or functions to optimize whereas the *Stacks* view helps you look at paths of execution that take a long time.

The project files view distinguishes between *Application Code* and *External Code*. You can choose which folders count as application code by right-clicking. *External Code* is typically system libraries that are hidden away at startup.

© 2015 Allinea Software Ltd.

22 MAP: Metrics View

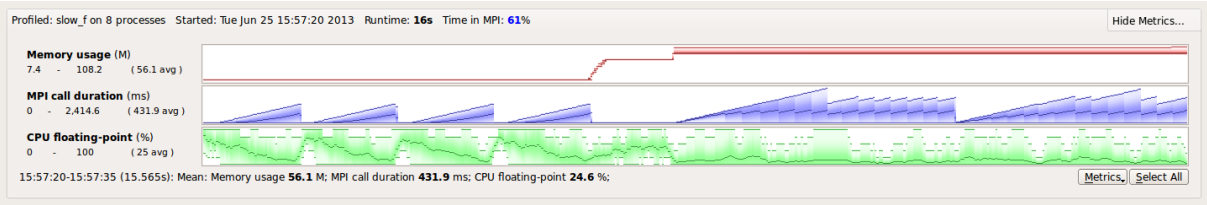

Profiled: slow_f on 8 processes Started: Tue Jun 25 15:57:20 2013 Runtime: 16s Time in MPI: 61%
Hide Metrics...
Memory usage (M)
7.4 - 108.2 (56.1 avg)
MPI call duration (ms)
0 - 2,414.6 (431.9 avg)
CPU floating-point (%)
0 - 100 (25 avg)
15:57:20-15:57:35 (15.565s): Mean: Memory usage 56.1 M; MPI call duration 431.9 ms; CPU floating-point 24.6 %;
Metrics
Select All

Figure 99: *Metrics view*

Now that you're familiar with the source code, the stacks and the project files view, let's see how the metrics view works with all three of them to help you identify, focus on and understand performance problems:

As with all graphs in MAP, the horizontal axis is wall clock time. By default three metric graphs are shown, displaying how each your application's use of memory, MPI call duration and floating-point (including SIMD instructions) varied across processes and time. Each vertical slice of a graph shows the distribution of values across processes for that moment in time - the minimum and maximum are clear, and shading is used to display the mean and standard deviation of the distribution.

A thin line means all processes had very similar values; a fat shaded region means there is significant imbalance between the processes. Extra details about each moment in time appear below the metric graphs as you move the mouse over them.

The metrics view is at the top of the GUI for a very good reason - it ties all the other views together. Move your mouse across one of the graphs, and a black vertical line will appear on every other graph in MAP, showing what was happening at that moment in time.

Even better, you can click and drag to select a region of time within it. All the other views and graphs now redraw themselves to show just what happened during the selected period of time, ignoring everything else. Try it and see! It's a fascinating way to isolate interesting parts of your application's execution. To re-select the entire time range just double-click or use the "Select All" button.

© 2015 Allinea Software Ltd.

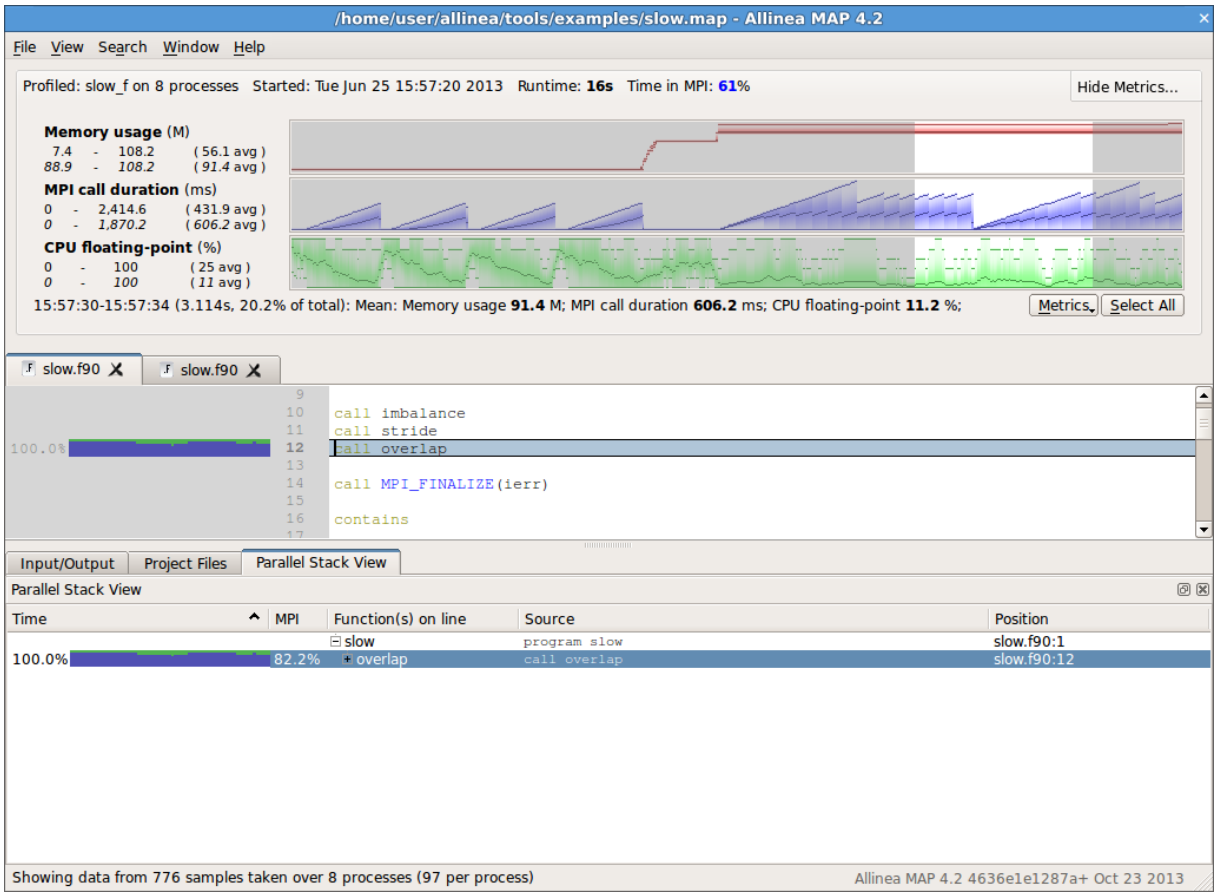


Figure 100: *Map with a region of time selected*

In the above screenshot a short region of time has been selected around an interesting sawtooth in time in `MPI_BARRIER` because PE 1 holds things up. The first block accepts data in PE order, so is badly delayed, the second block is more flexible, accepting data from any PE, so PE 1 can compute in parallel. The *Code View* shows how compute and comms are serialized in the first block, but overlap in the second.

There are many more metrics than those displayed by default. Click the *Metrics* button or right-click on the metric graphs and you can choose any combination of the following:

Memory Usage: The current RAM usage of each process. The interesting thing about this figure is that memory that is allocated and never used is generally not shown - only pages actively swapped into RAM by the OS count. This means that you'll often see memory usage ramp up as arrays are initialized. The slopes of these ramps can be interesting in themselves.

Note: this means if you `malloc` or `ALLOCATE` a large amount of memory but don't actually use it the Memory Usage *metric will not increase.*

MPI call duration: This metric tracks the time spent in an MPI call so far. PEs waiting at a barrier (MPI blocking sends, reductions, waits and barriers themselves) will ramp up time until finally they escape. Large areas show lots of wasted time and are prime targets for investigation. The pe with no time spent in calls is likely to be the last one to arrive, so should be the focus for any imbalance reduction.

MPI bytes sent / received: This pair of metrics tracks the number of bytes passed to MPI send / receive functions per second. This is not the same as the speed with which data is transmitted over the network - that information isn't available. This means that an MPI call that receives a large amount of data and completes almost instantly will have an unnaturally high instantaneous rate. These metrics may be replaced

in future releases - let us know if you want to keep them!

MPI point-to-point / collective operations: This pair of metrics tracks the number of point-to-point / collective calls per second. A long shallow period followed by a sudden spike is typical of a late sender - most processes are spending a long time in one MPI call (very low #calls per second) while one computes. When that one reaches the matching MPI call it completes much faster, causing a sudden spike in the graph. These metrics may be replaced in future releases - let us know if you want to keep them!

CPU floating-point: The percentage of time each rank spends in floating-point CPU instructions. This includes vectorized / SIMD instructions and standard x87 floating-point. High values here suggest CPU-bound areas of the code that are probably functioning as expected.

CPU integer: The percentage of time each rank spends in integer CPU instructions. This includes vectorized / SIMD instructions and standard integer operations. High values here suggest CPU-bound areas of the code that are probably functioning as expected.

CPU memory access: The percentage of time each rank spends in memory access CPU instructions, such as `move`, `load` and `store`. This also includes vectorized memory access functions. High values here may indicate inefficiently-structured code. Extremely high values (98% and above) almost always indicate cache problems. Typical cache problems include cache misses due to incorrect loop orderings but may also include more subtle features such as false sharing or cache line collisions.

CPU floating-point vector: The percentage of time each rank spends in vectorized / SIMD floating-point instructions. Well-optimized floating-point-based HPC code should spend most of its time running these operations; this metric provides a good check to see whether your compiler is correctly vectorizing hotspots. Future releases will show a breakdown between the kind of instructions e.g. SSE vs AVX. See section E.6 for a list of the instructions considered vectorized.

CPU integer vector: The percentage of time each rank spends in vectorized / SIMD integer instructions. Well-optimized integer-based HPC code should spend most of its time running these operations; this metric provides a good check to see whether your compiler is correctly vectorizing hotspots. See section E.6 for a list of the instructions considered vectorized.

CPU branch: The percentage of time each rank spends in test and branch-related instructions such as `test`, `cmp` and `je`. A well-optimized HPC code should not spend much time in branch-related instructions. Typically the only branch hotspots are during MPI calls, in which the MPI layer is checking whether a message has been fully-received or not. This metric may not be included in future releases - let us know if you find it useful.

Disk read transfer: The rate at which the application reads data from disk, in bytes per second. This includes data read from network filesystems (such as NFS), but may not include all local I/O due to page caching.

Disk write transfer: The rate at which the application writes data to disk, in bytes per second. This includes data written to network filesystems.

Note: Disk transfer metrics are not available on Cray X-series systems as the necessary Linux kernel support is not enabled.

You can return to the default set of metrics at any time by choosing the "Preset: Default" option.

22.1 Detecting MPI imbalance

The metrics view show the distribution of their value across all processes against time, so any 'fat' regions are showing an area of imbalance in this metric. Analysing imbalance in MAP works like this:

© 2015 Allinea Software Ltd.

1. Look at the metrics view for any 'fat' regions - these represent imbalance in that metric during that region of time. This tells us (A) that there is an imbalance, and (B) which metrics are affected.
2. Click and drag on the metrics view to select the 'fat' region, zooming the rest of the controls in to just this period of imbalance.
3. Now the stacks view and the source code views show which functions and lines of code were being executed during this imbalance. Are the processes executing different lines of code? Are they executing the same one, but with differing efficiencies? This tells us (C) which lines of code and execution paths are part of the imbalance.
4. Hover the mouse over the fattest areas on the metric graph and watch the minimum and maximum process ranks. This tells us (D) which ranks are most affected by the imbalance.

Now we know (A) whether there is an imbalance and (B) which metrics (CPU, memory, FPU, I/O) it affects. We also know (C) which lines of code and (D) which ranks to look at in more detail.

Often this is more than enough information to understand the immediate cause of the imbalance (e.g. late sender, workload imbalance) but for a deeper view we can now switch to DDT and re-run the program with a breakpoint in the affected region of code. Examining the two ranks highlighted as the minimum and maximum by MAP with the full power of an interactive debugger helps get to the root cause of the imbalance behaviour.

© 2015 Allinea Software Ltd.

23 Running MAP from the Command Line

MAP can be run from the command line with the following arguments:

-nompi

Run MAP with 1 process and without invoking `mpirun`, `mpiexec`, or equivalent.

-queue

Force MAP to submit the job to the queueing system.

-noqueue

Run MAP without submitting the job to the queueing system.

-profile

Generate a MAP profile but without user interaction. This will not display the MAP GUI. Messages are printed to the standard output and error. The job is not run using the queueing system unless used in conjunction with `-queue`. When the job finishes a map file is written and its name is printed.

23.1 Profiling MPMD Programs

The command to create a profile from an MPMD program using MAP is:

```
map <map mode> -n <#processes> -mpiargs <MPMD command> <one MPMD
   program>
```

This example shows how to run MAP without user interaction using the flag -profile:

```
map -profile -n 16 -mpiargs "-n 8 ./exe1 : -n 8 ./exe2" ./exe1
```

First we set the amount of processes used by the MPMD programs, in our case 8+8=16, then an MPMD style command as an mpi argument and finally one of the MPMD programs.

© 2015 Allinea Software Ltd.

24 Configuration

Note: DDT and MAP can share the same configuration files so in this section references to DDT can be replaced by MAP.

24.1 Configuration files

DDT and MAP are controlled by two configuration files: the system wide `system.config` and the user specific `user.config`. The system wide configuration file specifies properties such as MPI implementation. The user specific configuration file describes user's preferences such as font size. The files are controlled by environment variables:

Environment Variable	Default
`ALLINEA_TOOLS_USER_CONFIG`	`${ALLINEA_TOOLS_CONFIG_DIR}/user.config`
`ALLINEA_TOOLS_SYSTEM_CONFIG`	`${ALLINEA_TOOLS_CONFIG_DIR}/system.config`
`ALLINEA_TOOLS_CONFIG_DIR`	`${HOME}/.allinea`

24.1.1 Site Wide Configuration

If you are the system administrator, or have write-access to the installation directory, you can provide a configuration file which other users will be given a copy of – automatically – the first time that they start DDT/MAP.

This can save other users from the configuration process – which can be quite involved if site-specific configuration such as queue templates and job submission have to be crafted for your location.

First configure DDT/MAP normally and run a test program to make sure all the settings are correct. When you are happy with your configuration execute one of the following commands:

```
ddt -cleanconfig
```

```
map -cleanconfig
```

This will remove any user-specific settings (such as the last program you ran) from your configuration file to make a `system.config` file. Follow the instructions the command gives to make it available as a template for all users.

If you want to use DDT to attach to running jobs you will also need to create a file called `nodes` in the installation directory with a list of compute nodes you want to attach to. See section 4.8 Attaching To Running Programs for details.

24.1.2 Converting Legacy Site-Wide Configuration Files

If you have existing site-wide configuration files from a version of Allinea DDT prior to 4.0 you will need to convert them to the new 4.0 format. This can easily be done using the following command line:

```
ddt -config oldconfig.ddt -systemconfig newconfig.ddt -cleanconfig
```

Note: `newconfig.ddt` must not exist beforehand.

© 2015 Allinea Software Ltd.

24.1.3 Using Shared Home Directories on Multiple Systems

If your site uses the same home directory for multiple systems you may want to use a different configuration directory for each system.

You can do this by specifying the `ALLINEA_TOOLS_CONFIG_DIR` environment variable before starting DDT/MAP. For example if you use the `module` system you may choose to set `ALLINEA_TOOLS_CONFIG_DIR` according to which system the module was loaded on.

For example, say you have two systems: harvester with login nodes `harvester-login1` and `harvester-login2` and `sandworm` with login nodes `sandworm-login1` and `sandworm-login2`. You may add something like the snippet below to your module file:

```
case $(hostname) in
   harvester-login*)
     ALLINEA_TOOLS_CONFIG_DIR=$HOME/.allinea/harvester
     ;;
   sandworm-login*
     ALLINEA_TOOLS_CONFIG_DIR=$HOME/.allinea/sandworm
     ;;
esac
```

24.1.4 Using a Shared Installation on Multiple Systems

If you have multiple systems sharing a common Allinea Tools installation you may wish to have a different default configuration for each system. You can use the `ALLINEA_TOOLS_DEFAULT_SYSTEM_CONFIG` environment variable to specify a different file for each system. For example, you may add something like the snippet below to your `module` file:

```
case $(hostname) in
   harvester-login*)
     ALLINEA_TOOLS_DEFAULT_SYSTEM_CONFIG=/sw/allinea/tools/
        harvester.config
     ;;
   sandworm-login*
     ALLINEA_TOOLS_DEFAULT_SYSTEM_CONFIG=/sw/allinea/tools/
        sandworm.config
     ;;
esac
```

24.1.5 Importing Legacy Configuration

If you have used a version of Allinea DDT prior to 4.0 your existing configuration will be imported automatically. If the `DDTCONFIG` environment variable is set or the `-config` command-line argument used the existing configuration will be imported but legacy configuration file will not be modified, and subsequent configuration changes will be saved as described in the sections above.

© 2015 Allinea Software Ltd.

24.2 Integration With Queuing Systems

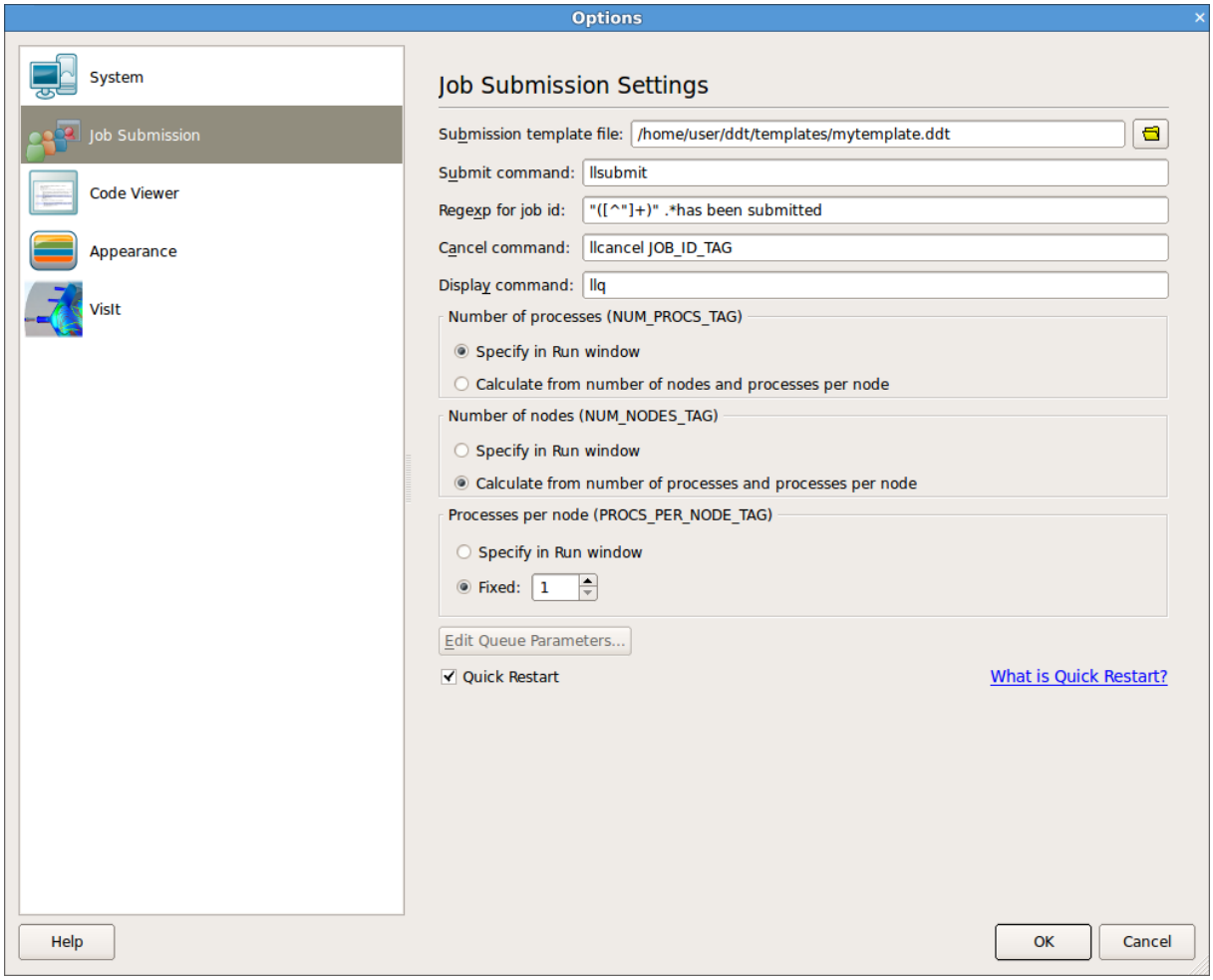


Figure 101: *Queuing Systems*

DDT/MAP can be configured to work with most job submission systems. In the *Options* window (Preferences on Mac OS X), you should choose *Submit job through queue*. This displays extra options and switches the software into queue submission mode.

The basic stages in configuring to work with a queue are:

1. Making a template script, and
2. Setting the commands used to submit, cancel, and list queue jobs.

Your system administrator may wish to provide a configuration file containing the correct settings, removing the need for individual users to configure their own settings and scripts.

In this mode, DDT/MAP uses a template script to interact with your queuing system. The `templates` subdirectory contains some example scripts that can be modified to meet your needs. `{installation-directory}/templates/sample.qtf`, demonstrates the process of creating a template file in some detail.

© 2015 Allinea Software Ltd.

24.3 Template Tutorial

Ordinarily, your queue script will probably end in a line that starts `mpirun` with your target executable. In most cases you can simply replace that line with `AUTO\_LAUNCH\_TAG`. For example, if your script currently has the line:

```
mpirun -np 16 program_name myarg1 myarg2
```

then create a copy of it and replace that line with:

```
AUTO_LAUNCH_TAG
```

Select this file as the *Submission template file* on the *Job Submission Settings* page of the *Options*. Notice that you are no longer explicitly specifying the number of processes, etc. You instead specify the number of processes, program name and arguments in the *Run* window.

Fill in *Submit command* with the command you usually use to submit your job, e.g. `qsub` or `sbatch`, *Cancel command* with the command you usually use to cancel a job, e.g. `qdel` or `scancel` and *Display command* with the command you usually use to display the current queue status, e.g. `qstat` or `squeue`.

You can usually use `(+)` as the *Regexp for job id* (this just looks for a number in the output from your *Submit command*).

Once you have a simple template working you can go on to make more things configurable from the GUI. For example, to be able to specify the number of nodes from the GUI you would replace an explicit number of nodes with the NUM_NODES_TAG, e.g. replace:

```
#SBATCH --nodes=100
```

with:

```
#SBATCH --nodes=NUM_NODE_TAG
```

See appendix F.1 Queue Template Tags for a full list of tags.

24.3.1 The Template Script

The template script is based on the file you would normally use to submit your job – typically a shell script that specifies the resources needed such as number of processes, output files, and executes `mpirun`, `vmirun`, `poe` or similar with your application.

The most important difference is that job-specific variables, such as number of processes, number of nodes and program arguments, are replaced by capitalized keyword tags, such as `NUM_PROCS_TAG`.

When DDT/MAP prepares your job, it replaces each of these keywords with its value and then submits the new file to your queue.

24.3.2 Configuring Queue Commands

Once you have selected a queue template file, enter submit, display and cancel commands.

When you start a session DDT/MAP will generate a submission file and append its file name to the submit command you give.

For example, if you normally submit a job by typing `job_submit -u myusername -f myfile` then you should enter `job_submit -u myusername -f` as the submit command.

© 2015 Allinea Software Ltd.

To cancel a job, DDT/MAP will use a regular expression you provide to get a value for JOB_ID_TAG. This tag is found by using regular expression matching on the output from your submit command. See appendix F.6 Job ID Regular Expression for details.

24.3.3 Configuring How Job Size is Chosen

DDT / MAP offer a number of flexible ways to specify the size of a job. You may choose whether *Number of Processes* and *Number of Nodes* options appear in the *Run* window or whether these should be implicitly calculated. Similarly you may choose to display *Processes per node* in the *Run* window or set it to a Fixed value.

Note: if you choose to display *Processes per node* in the Run window and PROCS_PER_NODE_TAG is specified in the queue template file then the tag will always be replaced by the *Processes per node* value from the *Run* dialog, even if the option is unchecked there.

24.3.4 Quick Restart

DDT allows you to quickly restart a job without resubmitting it to the queue if your MPI implementation supports it. Simply check the *Quick Restart* check box on the Job Submission Options page.

In order to use quick restart, your queue template file must use AUTO_LAUNCH_TAG to execute your job.

For more information on AUTO_LAUNCH_TAG, see F.4.1 Using AUTO_LAUNCH_TAG.

24.4 Connecting to remote programs (remote-exec)

When DDT needs to access another machine for remote launch or as part of starting some MPIs, it will attempt to use the secure shell, ssh, by default.[1]

However, this may not always be appropriate, ssh may be disabled or be running on a different port to the normal port 22. In this case, you can create a file called remote-exec which is placed in your ~/.allinea directory and DDT will use this instead.

DDT will use look for the script at ~/.allinea/remote-exec, and it will be executed as follows:

```
remote-exec HOSTNAME APPNAME [ARG1] [ARG2] ...
```

The script should start APPNAME on HOSTNAME with the arguments ARG1 ARG2 without further input (no password prompts). Standard output from APPNAME should appear on the standard output of remote-exec. An example is shown below:

SSH based remote-exec

A remote-exec script using ssh running on a non-standard port could look as follows:

```
#!/bin/sh
ssh -P {port-number} $*
```

In order for this to work without prompting for a password, you should generate a public and private SSH key, and ensure that the public key has been added to the ~/.ssh/authorized_keys file on machines you wish to use. See the ssh-keygen manual page for more information.

[1] The remote-exec script is not used on Windows, so this section is inapplicable to that platform.

Testing

Once you have set up your `remote-exec` script, it is recommended that you test it from the command line. e.g.

```
~/.allinea/remote-exec TESTHOST uname -n
```

Should return the output of `uname -n` on `TESTHOST`, without prompting for a password.

If you are having trouble setting up `remote-exec`, please contact support@allinea.com for assistance.

24.5 Optional Configuration

DDT/MAP provides an options window (Preferences on Mac OS X), which allows you to quickly edit the settings in the configuration wizard, as well as other non-essential preferences. These options are outlined briefly below.

24.5.1 System

MPI Implementation: Allows you to tell DDT/MAP which MPI implementation you are using.

Note: If you are not using DDT to debug MPI programs select `none`.

Override default mpirun path: Allows you to override the path to the `mpirun`(or equivalent) command.

Select Debugger: Tells DDT/MAP which underlying debugger it should use. This should almost always be left as *Automatic*.

On Linux systems DDT 4.2.1 ships with two versions of the GNU GDB debugger: GDB 7.2 and GDB 7.6.2. GDB 7.2 is the same version as ships with DDT 4.2 and is provided for backwards compatibility. GDB 7.6.2 provides, amongst other things, improved DWARF 4 and C++ support and is recommended if you are using a recent compiler such as GCC 4.8.

Create Root and Workers groups automatically: If this option is checked DDT will automatically create a *Root* group for rank 0 and a *Workers* group for ranks 1... n when you start a new MPI session.

Use Shared Symbol Cache: The shared symbol cache is a file that contains all the symbols in your program in a format that can be used directly by the debugger. Rather than loading and converting the symbols itself, every debugger shares the same cache file. This significantly reduces the amount of memory used on each node by the debuggers. For large programs there may be a delay starting a job while the cache file is created as it may be quite large. The cache files are stored in `$HOME/.allinea/symbols`. We recommend only turning this option on if you are running out of memory on compute nodes when debugging programs with DDT.

Heterogeneous system support: DDT has support for running heterogeneous MPMD MPI applications where some nodes use one architecture and other nodes use another architecture. This requires a little preparation of your DDT installation. You must have a separate installation of DDT for each architecture. The architecture of the machine running the DDT GUI is called the host architecture. You must create symbolic links from the host architecture installation of DDT to the other installations for the other architectures. For example with a 64-bit x86_64 host architecture (running the GUI) and some compute nodes running the 32-bit i686 architecture:

```
ln -s /path/to/ddt-i686/bin/ddt-debugger \
/path/to/ddt-x86_64/bin/ddt-debugger.i686
```

© 2015 Allinea Software Ltd.

Enable CUDA software pre-emption (CUDA 5.5+): Allows debugging of CUDA kernels on a workstation with a single GPU.

Copy files to compute nodes
Xeon Phi: Copy the DDT debugger daemon files to the Xeon Phi cards when attaching to an offload process. With this option enabled the DDT installation does not need to be visible on the Phi card - i.e. no shared filesystem is required.
Bluegene: Copy the DDT debugger daemon files to the Bluegene I/O nodes. This may offer better debugging performance, but comes at the expense of consuming more RAM on the I/O nodes. To use this option approximately 50% of the I/O node RAM should be free during normal operation, otherwise you risk exhausting the RAM on the I/O nodes.

Default groups file: Entering a file here allows you to customise the groups displayed by DDT when starting an MPI job. If you do not specify a file DDT will create the default *Root* and *Workers* groups if the previous option is checked.

Note: A groups file can be created by right clicking the process groups panel and selecting Save groups... while running your program.

Attach hosts file: When attaching, DDT will fetch a list of processes for each of the hosts listed in this file. See section 4.8 Attaching To Running Programs for more details.

24.5.2 Job Submission

This section allows you to configure DDT/MAP to use a custom `mpirun`command, or submit your jobs to a queuing system. For more information on this, see section 24.2 Integration With Queuing Systems.

24.5.3 Code Viewer Settings

This allows you to configure the appearance of the DDT/MAP code viewer (used to display your source code while debugging)

Tab size : Sets the width of a tab character in the source code display. (A width of 8 means that a tab character will have the same width as 8 space characters.)

Font name: The name of the font used to display your source code. It is recommended that you use a fixed width font.

Font size: The size of the font used to display your source code.

Editor : This is the program DDT/MAP will execute if you right click the code viewer and choose *Open file in editor*. This command should launch a graphical editor. If no editor is specified, DDT/MAP will attempt to launch a default editor (as configured in your desktop environment).

Colour Scheme: Colour palette to use for the code viewer's background, text and syntax highlighting. Defined in Kate syntax definition format in the `resource/styles` directory of the DDT/MAP install.

Visualize Whitespace: Enables or disables this display of sybols to represent whitespace. Useful for distinguishing between space and tab characters.

Warn about potential programming errors: This setting enables or disables the use of static analysis tools that are included with the Allinea DDT installation. These tools support F77, C and C++, and analyse the source code of viewed source files to discover common errors, but can cause heavy CPU usage on the system running the DDT user interface. You can disable this by unchecking this option.

© 2015 Allinea Software Ltd.

24.5.4 Appearance

This section allows you to configure the graphical style of DDT/MAP, as well fonts and tab settings for the code viewer.

Look and Feel: This determines the general graphical style of DDT/MAP. This includes the appearance of buttons, context menus.

Override System Font Settings: This setting can be used to change the font and size of all components in DDT/MAP (except the code viewer).

24.5.5 VisIt

Allow the use of VisIt with DDT: When ticked DDT will launch VisIt whenever a *Vispoint* is hit, or on demand.

VisIt launch command: The full path to the VisIt binary (`visit`).

Custom Arguments: Any extra arguments to pass to VisIt.

Launch VisIt with small viewer: Launches VisIt with a smaller viewer window.

Use Hardware Acceleration: Enable hardware acceleration (uses GPU for rendering).

Raise DDT window when a 'DDT pick' is made in VisIt: When enabled, selecting an cell/zone with the DDT pick tool within VisIt will cause DDT to attempt to raise its window to the top of your desktop. Note that this may not be successful as many window managers prevent applications from raising themselves in this way.

Close VisIt when the DDT session ends: It is not possible to interact with a VisIt visualization once DDT has ended the session (the program containing the arrays to visualize no longer exists). To avoid confusion and prevent problems when next viewing a VisIt visualization it is recommended VisIt be closed when the DDT session ends, and a fresh VisIt instance launched as needed for the next visualization.

Enable vispoints: When a visualization breakpoint is hit DDT transfers control to VisIt where you can visualise a given array from your program. Vispoints do not work with programs that are already instrumented for use with VisIt (See section 16.9 Using DDT with a pre-instrumented program).

Automatically create Pseudocolour plots for variables: When a new vispoint is hit the vispoint's array variables will automatically be plotted (using pseudocolour plots) within VisIt. When there are multiple array variables to visualize each will be plotted in a viewer window. Note that this feature uses the VisIt CLI and that whilst the viewer windows are being created and configured a CLI terminal window will be briefly visible.

If you see a dialog “VisIt is waiting for a CLI to launch” for more than 10 seconds it is likely VisIt is unable to provide a CLI on your system. On Linux systems the program `xterm` is required to use VisIt's CLI.

VisIt launch command on compute nodes: The full path to the VisIt binary on the compute nodes (if different from the frontend).

© 2015 Allinea Software Ltd.

25 The Licence Server

The licence server supplied with DDT/MAP is capable of serving clients for several different licences, enabling one server to serve all Allinea software in an organization. There is no need to run separate licence servers for DDT and MAP – one combined licence server provides licences for both products.

25.1 Running The Server

For security, the licence server should be run as an unprivileged user (e.g. `nobody`). If run without arguments, the server will use licences in the current directory (files matching `Licence*` and `License*`). An optional argument specifies the pathname to be used instead of the current.

System administrators will normally wish to add scripts to start the server automatically during booting. `allinea_licensing_init` is a SysV init style script for Red Hat Enterprise Linux and SuSE Enterprise Linux that may be used to start the server. To install the script follow the instructions below as root:

1. Edit `ALLINEA_TOOLS_PATH` at the top of the script to point to the Allinea tools installation.
2. Set `ALLINEA_TOOLS_PATH` to the path to the Allinea tools installation.
3. Copy the script to `/etc/init.d`:

```
cp "${ALLINEA_TOOLS_PATH}/bin/allinea_licensing_init" /etc/
   init.d/
```

4. Make a new user to run the licenceserver:

```
adduser --system --user-group --no-create-home --home-dir "
   ${ALLINEA_TOOLS_PATH}/tools" allinea
```

5. Create the log files directory:

```
mkdir /var/log/allinea
chown allinea.allinea /var/log/allinea
```

6. Enable the service at boot time:

```
chkconfig --add allinea_licensing_init
```

7. Start the service:

```
service allinea_licensing_init start
```

8. Check that is started ok:

```
service allinea_licensing_init status
```

Logs may be found in `/var/log/allinea.log` by default.

25.2 Running DDT/MAP Clients

DDT/MAP will, as is also the case for fixed licences, use a licence directory either specified via environment variables (`ALLINEA_LICENCE_DIR` or `ALLINEA_LICENSE_DIR`) or from the default location of `{installation-directory}/licences`.

© 2015 Allinea Software Ltd.

In the case of floating licences this file is unverified and in plain-text, it can therefore be changed by the user if settings need to be amended.

The fields are:

Name	Required	Description
hostname	Yes	The hostname, or IP address of the licence server
ports	No	A comma separated list of ports to be tried locally for GUI-backend communication, Defaults to 4242,4243,4244,4244,4245
serial_number	Yes	The serial number of the server licence to be used
serverport	Yes	The port the server listens on
type	Yes	Must have value 2 – this identifies the licence as needing a server to run properly

Note: The serial number of the server licence is specified as this enables a user to be tied to a particular licence.

25.3 Logging

Set the environment variable `ALLINEA_LICENCE_LOGFILE` to the file that you wish to append log information to. Set `ALLINEA_LICENCE_LOGLEVEL` to set the amount of information required. These steps must be done prior to starting the server.

- Level 0: no logging.
- Level 1: client licences issued are shown, served licences are listed.
- Level 2: stale licences are shown when removed, licences still being served are listed if there is no spare licence.
- Level 3: full request strings received are displayed
- Level 6: is the maximum.

In level 1 and above, the MAC address, user name, process ID, and IP address of the clients are logged.

25.4 Troubleshooting

Licences are plain-text which enables the user to see the parameters that are set; a checksum verifies the validity. If problems arise, the first step should be to check the parameters are consistent with the machine that is being used (MAC and IP address), and that, for example, the number of users is as expected.

25.5 Adding A New Licence

To add a new licence to be served, copy the file to the directory where the existing licences are served and restart the server. Existing clients should not experience disruption, if the restart is completed within a minute or two.

© 2015 Allinea Software Ltd.

25.6 Examples

In this example, a dedicated licence server machine exists but uses the same file system as the client machines, and the software is installed at `/opt/allinea/tools`.

To run the `licenceserver` as nobody, serving all licences in `/opt/allinea/tools`, and logging most events to the `/var/log/allinea.log`.

```
% su – nobody
Password:
% export ALLINEA_LICENCE_LOGFILE=/var/log/allinea.log
% export ALLINEA_LICENCE_LOGLEVEL=2
% cd /opt/allinea/tools
% ./bin/licenceserver /opt/allinea/tools/ &
% exit
```

Serving the floating licences from the same directory as a normal installation is possible as the licence server will ignore licences that are not server licences.

If the server licence is file `/opt/allinea/tools/licences/Licence.server.physics` and is served by the machine `server.physics.acme.edu`, on port 4252, the licence would look like:

```
type=3
serial_number=1014
max_processes=48
expires=2004-04-01 00:00:00
support_expires=2004-04-01 00:00:00
mac=00:E0:81:03:6C:DB
interface=eth0
debuggers=gdb
serverport=4252
max_users=2
beat=60
retry_limit=4
hash=P5I:?L,FS=[CCTB<IW4
hash2=c18101680ae9f8863266d4aa7544de58562ea858
```

Then the client licence could be stored at `/opt/allinea/tools/licences/Licence.client.physics` and contain:

```
type=2
serial_number=1014
hostname=server.physics.acme.edu
serverport=4252
```

25.7 Example Of Access Via A Firewall

SSH forwarding can be used to reach machines that are beyond a firewall, for example the remote user would start:

```
ssh -C -L 4252:server.physics.acme.edu:4242 login.physics.acme.edu
```

And a local licence file should be created:

```
type=2
serial_number=1014
hostname=localhost
serverport=4252
```

25.8 Querying Current Licence Server Status

The licence server provides a simple HTML interface to allow for querying of the current state of the licences being served. This can be accessed in a web browser at the following URL:

```
http://<hostname>:<serverport>/status.html
```

For example, using the values from the licence file examples, above:

```
http://server.physics.acme.edu:4252/status.html
```

Initially, no licences will be being served, and the output in your browser window should look something like:

```
[Licences start]
  [Licence Serial Number: 1014]
    [No licences allocated - 2 available]
[Licences end]
```

As licences are served and released, this information will change. To update the licence server status display, simply refresh your web browser window. For example, after one DDT/MAP has been started:

```
[Licences start]
  [Licence Serial Number: 1014]
    [1 licences available]
      [Client 1]
        [mac=00:04:23:99:79:65; uname=gwh; pid=14007; licence=1014]
        [Latest heartbeat: 2004-04-13 11:59:15]
[Licences end]
```

Then, after another DDT/MAP is started and the web browser window is refreshed (notice the value for number of licences available):

```
[Licences start]
  [Licence Serial Number: 1014]
    [0 licences available]
      [Client 1]
        [mac=00:04:23:99:79:65; uname=gwh; pid=14007; licence=1014]
        [Latest heartbeat: 2004-04-13 12:04:15]
      [Client 2]
        [mac=00:40:F4:6C:4A:71; uname=graham; pid=3700; licence
           =1014]
        [Latest heartbeat: 2004-04-13 12:04:59]
[Licences end]
```

Finally, after the first DDT/MAP finishes:

© 2015 Allinea Software Ltd.

```
[Licences start]
  [Licence Serial Number: 1014]
    [1 licences available]
      [Client 1]
        [mac=00:40:F4:6C:4A:71; uname=graham; pid=3700; licence
           =1014]
        [Latest heartbeat: 2004-04-13 12:07:59]
[Licences end]
```

25.9 Licence Server Handling Of Lost DDT/MAP Clients

Should the licence server lose communication with a particular instance of a client, the licence allocated to that particular client will be made unavailable for new clients until a certain time out period has expired. The length of this time out period can be calculated from the licence server file values for `beat` and `retry_limit`:

```
lost_client_timeout_period = (beat seconds) * (retry_limit + 1)
```

So, for the example licence files above, the time out period would be:

```
60 * (4 + 1) = 300 seconds
```

During this time out period, details of the 'lost' client will continue to be output by the licence server status display. As long as additional licences are available, new clients can be started. However, once all of these additional licences have been allocated, new clients will be refused a licence while this time out period is active.

After this time out period has expired, the licence server status will continue to display details of the 'lost' client until another client is started. The licence server will grant a licence to the new client and the licence server status display will then reflect the details of the new client.

© 2015 Allinea Software Ltd.

A Supported Platforms

A full list of supported platforms and configurations is maintained on the Allinea website. It is likely that MPI distributions supported on one platform will work immediately on other platforms.

A.1 DDT

See http://www.allinea.com/products/ddt-platforms.

Platform	Operating Systems	MPI	Compilers
x86 and x86_64	Red Hat Enterprise Linux 5, 6 and derivatives, SLES 11 and above, Ubuntu 12.04 and above	*All known MPIs – including but not limited to:* All known MPI implementations and platforms – including but not limited to: SGI Altix , Bproc, Cray, Bull MPI 1 and 2, LAM-MPI, MPICH 1, MPICH 2, MPICH 3, Myricom MPICH-GM and MPICH-MX, Open MPI, Quadrics MPI, Platform (Scali) MPI, Scyld, Intel MPI, MVAPICH 1, MVAPICH 2	GNU, Absoft, Cray, Intel, Pathscale, PGI, Oracle
Intel Xeon Phi (Intel MIC)	MPSS 2.1.4982-15, 2.1.6720-19, 3.1	Intel MPI and native mode	Intel, GNU
IBM Power	AIX 5.3, 6.0 and 6.1, Red Hat Enterprise Linux 6	IBM PE, MPICH, Open MPI	IBM XLC, IBM XLF, GNU
Blue Gene/Q	Red Hat Enterprise Linux 6	Native	GNU and IBM
ARM v7	Ubuntu 12.04, Other embedded devices via gdbserver	*All*	GNU
NVIDIA CUDA Toolkit 4.2/5.0/5.5/6.0/6.5	Linux	*All*	CAPS HMPP, Cray OpenMP Accelerators, NVCC, PGI Accelerators, PGI CUDA Fortran

Batch scheduling systems such as SLURM, PBS, TORQUE, Moab, Oracle Grid Engine and Loadleveler are supported through Queue Templates - see section 24.2 Integration With Queuing Systems for more information. See section B.13 SLURM for more details about SLURM support.

© 2015 Allinea Software Ltd.

A.2 MAP

See http://www.allinea.com/products/map/platforms.

Platform	Operating Systems	MPI	Compilers
x86_64	Redhat Enterprise Linux 5, 6 and derivatives, SLES 11, Ubuntu 12.04 and above	MVAPICH 1, MPICH 2, MPICH 3, Open MPI, Intel MPI, Cray MPT, Slurm, SGI MPT MPICH 1 is **not** supported.	GNU, Intel, Cray, PGI Other compilers supporting the DWARF standard e.g. Pathscale should also work.
Intel Xeon Phi (Intel MIC)	MPSS 2.1.6720-19, 3.1 Please note: older versions of MPSS are **not** supported.	Intel MPI and native mode	Intel, GNU

Batch scheduling systems such as SLURM, PBS, TORQUE, Moab, Oracle Grid Engine and Loadleveler are supported through Queue Templates - see section 24.2 Integration With Queuing Systems for more information. See section B.13 SLURM for more details about SLURM support.

© 2015 Allinea Software Ltd.

B MPI Distribution Notes and Known Issues

This appendix has brief notes on many of the MPI distributions supported by DDT. Advice on settings and problems particular to a distribution are given here. MAP, as a newer product, has only been tested on a subset of these, but may work on the others.

B.1 Bull MPI

Bull MPI 1, MPI 2 and Bull X-MPI are supported. For Bull X-MPI select the Open MPI or Open MPI (Compatibility) MPIs, depending on whether ssh is allowed (in which case choose Open MPI) or not (choose Open MPI Compatibility mode).

Select *Bull MPI or Bull MPI 1 for* Bull MPI 1, *or Bull MPI 2* for Bull MPI 2 from the MPI implementations list. In the *mpirun arguments* box of the *Run* window you may also wish to specify the partition that you wish to use – by adding

```
-p partition_name
```

You should ensure that `prun`, the command used to launch jobs, is in your `PATH` before starting DDT.

B.2 HP MPI

Select *HP MPI* as the MPI implementation.

A number of HP MPI users have reported a preference to using `mpirun -f jobconfigfile` instead of `mpirun -np 10 a.out` for their particular system. It is possible to configure DDT to support this configuration – using the support for batch (queuing) systems.

The role of the queue template file is analogous to the `-f jobconfigfile`.

If your job config file normally contains:

```
-h node01 -np 2 a.out
-h node02 -np 2 a.out
```

Then your template file should contain:

```
-h node01 -np PROCS_PER_NODE_TAG /usr/local/ddt/bin/ddt-debugger
-h node02 -np PROCS_PER_NODE_TAG /usr/local/ddt/bin/ddt-debugger
```

and the *Submit Command* box should be filled with

```
mpirun -f
```

Select the *Template uses `NUM_NODES_TAG` and `PROCS_PER_NODE_TAG`* radio button. After this has been configured by clicking `OK`, you will be able to start jobs. Note that the *Run* button is replaced with *Submit*, and that the number of processes box is replaced by *Number of Nodes*.

B.3 Intel MPI

Select *Intel MPI* from the MPI implementation list. DDT has been tested with Intel MPI 2.0 onwards.

DDT also supports the Intel Message Checker tool that is included in the Intel Trace Analyser and Collector software. A plugin for the Intel Trace Analyser and Collector version 7.1 is provided in DDT's

plugins directory. Once you have installed the Intel Trace Analyser and Collector, you should make sure that the following directories are in your `LD_LIBRARY_PATH`:

```
{path to intel install directory}/itac/7.1/lib
{path to intel install directory}/itac/7.1/slib
```

The Intel Message Checker only works if you are using the Intel MPI. Make sure Intel's mpiexec is in your path, and that your application was compiled against Intel's MPI, then launch DDT, check the plugin checkbox and debug your application as usual. If one of the above steps has been missed out, DDT may report an error and say that the plugin could not be loaded.

Once you are debugging with the plugin loaded, DDT will automatically pause the application whenever Intel Message Checker detects an error. The Intel Message Checker log can be seen in the standard error (stderr) window.

Note that the Intel Message Checker will abort the job after 1 error by default. You can modify this by adding `-genv VT_CHECK_MAX_ERRORS 0` to the *mpiun arguments* box in the *Run* window – see Intel's documentation for more details on this and other environment variable modifiers.

B.4 MPICH 2

If you see the error `undefined reference to MPI_Status_c2f` while building the MAP libraries (17.1.3 Linking) then you need to rebuild MPICH 2 with Fortran support.

B.5 MPICH 3

MPICH 3.0.3 and 3.0.4 do not work with Allinea DDT or Allinea MAP due to a defect. MPICH 3.1 addresses this and is supported.

There are two MPICH 3 modes – standard and Compatibility. If the standard mode does not work on your system select *MPICH 3 (Compatibility)* as the *MPI Implementation* on the *System Settings* page of the *Options* window.

B.6 IBM PE

Ensure that `poe` is in your path.

A sample Loadleveler script, which starts debugging jobs on IBM AIX (POE) systems is included in the {*installation-directory*}`/templates` directory.

On AIX 5.3 TL12 when working via Loadleveler, some users have experienced a POE imposed process count limit and been unable to debug above 5 MPI processes per node. This a known IBM issue and the default queue script for DDT and the `ddt-client` script used by it contains a workaround. MAP does not support AIX.

Select *IBM PE* as the MPI implementation.

B.7 MVAPICH 1

You will need to specify the hosts on which to launch your job to mvapich's `mpirun`by using the `-hostfile filename` or individually as per the MVAPICH documentation in the *mpirun Arguments* box.

See section 17.1.4 Static Linking for additional notes on linking the MAP MPI wrapper with MVAPICH 1.

B.8 MVAPICH 2

Known issue: If memory debugging is enabled in DDT, this will interfere with the on-demand connection system used by MVAPICH2 above a threshold process count and applications will fail to start. This threshold default value is 64. To work around this issue, set the environment variable `MV2_ON_DEMAND_THRESHOLD` to the maximum job size you expect on your system and then DDT will work with memory debugging enabled for all jobs. This setting should not be a system wide default as it may increase startup times for jobs and memory consumption.

MVAPICH 2 now offers `mpirun_rsh` instead of `mpirun`as a scalable launcher binary – to use this with DDT, from *File* → *Options* (*DDT* → *Preferences* on Mac OS X) go to the *System* page, check *Override default mpirun path* and enter `mpirun_rsh`.

B.9 Open MPI

DDT has been tested with Open MPI 1.2.x 1.3.x, 1.4.x, 1.6.x and 1.8.x. Select *Open MPI* from the list of MPI implementations.

There are three different Open MPI choices in the list of MPI implementations to choose from in DDT when debugging for Open MPI.

- *Open MPI* - the job is launched with a custom 'launch agent' that, in turn, launches the Allinea daemons.
- *Open MPI (Compatibility)* - mpirun launches the Allinea daemons directly. This startup method does not take advanatage of Allinea's scalable tree.
- *Open MPI for Cray XT/XE/XK/XC* - for Open MPI running on Cray XT/XE/XK/XC systems. This method is fully able to use DDT's scalable tree infrastructure for large scale debugging.

Known issue: If you are using the 1.6.x series of Open MPI configured with the `--enable-orterun-prefix-by-default` flag then DDT requires patch release 1.6.3 or later due to a defect in earlier versions of the 1.6.x series.

Known issue: The version of Open MPI packaged with Ubuntu has the Open MPI debug libraries stripped. This prevents the *Message Queues* feature of DDT from working.

Known issue: With Open MPI 1.3.4 and Intel Compiler v11 – the default build will optimize away a vital call during the startup protocol which means the default Open MPI start up will not work. If this is your combination, either update your Open MPI, or select *Open MPI (Compatibility)* instead as the DDT MPI Implementation.

Known Issue: On Infiniband systems, Open MPI and CUDA can conflict in a manner that results in failure to start processes, or a failure for processes to be debuggable. CUDA 4.0 and above can co-exist with Infiniband – but requires that the environment variable `CUDA_NIC_INTEROP` is set to 1. An alternative for CUDA 3.2 is to disable Infiniband completely (provide `-mca btl ^openib` as an extra mpirun argument in the DDT run dialog). For more information of this issue please see: http://cudamusing.blogspot.co.uk/2011/08/cuda-mpi-and-infiniband.html.

© 2015 Allinea Software Ltd.

B.10 SGI Altix / SGI MPT

If using SGI MPT 2.08+, select SGI MPT (2.08+, batch) as the MPI implementation.
If using an older version of SGI MPT (2.07 or before) select SGI MPT as the MPI implementation. Note that support for SGI MPT 2.10+ scalable startup was removed in version 4.2.1-38188 and will be re-added in a future release.

If you are using SGI MPT with PBS or SLURM and would normally use `mpiexec_mpt` to launch your program you will need to use the `pbs-sgi-mpt.qtf` queue template file and select `SGI MPT` (`Batch`) as the MPI implementation.

`mpiexec_mpt` from versions of SGI MPT prior to 2.10 may prevent MAP from starting when preloading the MAP sampler and MPI wrapper. We recommend you explicitly link your programs against the MAP libraries to work around this problem.

SGI MPT 2.09 requires the `MPI_SUPPORT_DDT` environment variable to be set to `1` to avoid startup issues when debugging with DDT.

B.11 Cray MPT

This section only applies when using `aprun`. For `srun` ('Native' SLURM mode) see section B.13 SLURM.

DDT has been tested with Cray XT 5/6, XE6, XK6/7, and XC30 systems – with DDT submitting via the queue and also from within an interactive shell. DDT is able to launch and support debugging jobs in excess of 700,000 cores.

A number of template files for launching applications from within the queue (using DDT's job submission interface) are included in the distribution of DDT – these may require some minor editing to cope with local differences on your batch system.

To attach to a running job on a Cray system the MOM nodes (those nodes where aprun is launched) must be reachable via ssh from the node where DDT is running (eg. a login node). DDT must connect to these nodes in order to launch debugging daemons on the compute nodes. Users can either specify the aprun-host manually in the attach dialog when scanning for jobs, or configure a hosts list containing all MOM nodes.

By default attempting to preload the memory debugging library will raise an error - the library must be explicitly linked with your program. You can disable the error by setting the `DDT_ALLOW_CRAY_DMALLOC_PRELOAD` to `1` before starting DDT. Preloading requires aprun/ALPS 4.1 or later and your program must be dynamically linked.

B.11.1 Using DDT with Cray ATP (the Abnormal Termination Process)

DDT is compatible with the Cray ATP system, which will be default on some XE systems. This runtime addition to applications automatically gathers crashing process stacks, and can be used to let DDT attach to a job before it is cleaned up during a crash.

To be able to debug after a crash when an application is run with ATP but without a debugger, the `ATP_HOLD_TIME` environment variable should be initialized before launching the job – a value of 5 is (very) ample, even on a large Petscale system, giving 5 minutes for the attach to complete.

The following example shows the typical output of an ATP session.

```
n10888@kaibab:~> aprun -n 1200 ./atploop
Application 1110443 is crashing. ATP analysis proceeding...
Stack walkback for Rank 23 starting:
 _start@start.S:113
 __libc_start_main@libc-start.c:220
 main@atploop.c:48
 __kill@0x4b5be7
Stack walkback for Rank 23 done
Process died with signal 11: 'Segmentation fault'
View application merged backtrace tree file 'atpMergedBT.dot'
  with 'statview'
You may need to 'module load stat'.

atpFrontend: Waiting 5 minutes for debugger to attach...
```

At this point, DDT can be launched to debug the application.

DDT can attach using the Attaching dialogs described in Section 4.8 Attaching To Running Programs, or given the PID of the aprun process, the debugging set can be specified from the command line.

For example, to attach to the entire job:

```
ddt --attach-mpi 12772
```

If a particular subset of processes are required, then the subset notation could also be used to select particular ranks.

```
ddt –attach-mpi 12772 -subset 23,100-112,782,1199
```

B.12 Berkeley UPC

Only the MPI transport is supported. Programs must be compiled with the `-tv` flag, e.g.

```
upcc hello.c -o hello -g -tv
```

B.13 SLURM

To start MPI programs using the `srun` command instead of your MPI's usual `mpirun` command (or equivalent) select *SLURM (MPMD)* as the *MPI Implementation* on the *System Settings* page of the *Options*. Note: this option will work with most MPIs, but not all. See below for some common exceptions.

Exceptions:

- On the Cray, 'Hybrid' SLURM mode (i.e. `sbatch` + `aprun`) is not supported - you must start your program with Cray's `aprun` instead. See Section B.11 Cray MPT.
- Bluegene/Q users should select *Bluegene/Q (SLURM)* as the *MPI Implementation* instead.

SLURM may be used as a job scheduler with DDT and MAP through the use of a queue template file - see `templates/slurm.qtf` in the Allinea tools installation for an example and section 24.2 Integration With Queuing Systems for more information on how to customize the template.

C Compiler Notes and Known Issues

When compiling for a DDT debugging session always compile with a minimal amount of, or no, optimization - some compilers reorder instruction execution and omit debug information when compiled with optimization turned on.

C.1 AMD OpenCL compiler

Not supported by MAP.

The AMD OpenCL compiler can produce debuggable OpenCL binaries – however, the target must be the CPU rather than the GPU device. The build flags `-g -O0` must be used when building the OpenCL kernel, typically by setting the environment variable:

```
AMD_OCL_BUILD_OPTIONS_APPEND="-g -O0"
```

The example codes in the AMD OpenCL toolkit are able to run on the CPU by adding a parameter `--device cpu` - and will result, with the above environment variable set, in debuggable OpenCL.

C.2 Berkeley UPC Compiler

Not supported by MAP.

The Berkeley UPC compiler is fully supported by Allinea DDT, but only when using the MPI conduit (other conduits are not supported).

Warning: If you don't compile the program fixing the number of threads (using the -fupc-threads-<numberOfThreads> flag), a known issue arises at the end of the program execution.

Note: Source files must end with the extension `.upc` in order for UPC support to be enabled.

C.3 Cray Compiler Environment

DDT supports Cray Fast Track Debugging. In DDT 4.2.1 it is only supported when using GDB 7.2 and not when using GDB 7.6.2. To enable fast track debugging compile your program with `-Gfast` instead of `-g`. See the *Using Cray Fast-track Debugging* section of the *Cray Programming Environment User's Guide* for more information.

Call-frame information can also be incorrectly recorded, which can sometimes lead to DDT stepping into a function instead of stepping over it. This may also result in time being allocated to incorrect functions in MAP.

C++ pretty printing of the STL is not supported by DDT for the Cray compiler.

See CUDA/GPU debugging notes for details of Cray OpenMP Accelerator support.

Allinea DDT fully supports the Cray UPC compiler. Not supported by MAP.

C.4 GNU

The compiler flag `-fomit-frame-pointer` should never be used in an application which you intend to debug/profile. Doing so can mean DDT/MAP cannot properly discover your stack frames and you will be unable to see which lines of code your program has stopped at.

© 2015 Allinea Software Ltd.

For GNU C++, large projects can often result in vast debug information size, which can lead to large memory usage by DDT's back end debuggers – for example each instance of an STL class used in different object files will result in the compiler generating the same information in each object file.

C.4.1 GNU UPC

DDT also supports the GCC-UPC compiler (`upc_threads_model_process` only; the `pthread-tls` threads model is not supported). MAP does not support this.

To compile and install GCC UPC 4.8 without TLS it is necessary to modify the configuration file `path/to/upc/source/code/directory/libgupc/configure`, replacing all the entries *upc_cv_gcc_tls_supported=``yes''* to *upc_cv_gcc_tls_supported=``no''*.

To run a UPC program in DDT you have to select the MPI implementation ``GCC libupc SMP (no TLS)''

C.5 IBM XLC/XLF

It is advisable to use the `-qfullpath` option to the IBM compilers (XLC/XLF) in order for source files to be found automatically when they are in directories other than that containing the executable. This flag has been known to fail for mpxlf95, and so there may be circumstances when you must right click in the project navigator and add additional paths to scan for source files.

Module data items behave differently between 32 and 64 bit mode, with 32-bit mode generally enabling access to more module variables than 64-bit mode.

Missing debug information in the binaries produced by XLF can prevent DDT from showing the values in Fortran pointers and allocatable arrays correctly, and assumed-size arrays cannot be shown at all. Please update to the latest compiler version before reporting this to support@allinea.com .

Sometimes, when a process is paused inside a system or library call, DDT will be unable to display the stack, or the position of the program in the Code view. To get around this, it is sometimes necessary to select a known line of code and choose *Run to here*. If this bug affects you, please contact support@allinea.com .

OpenMP loop variables are often optimized away and not present when debugging.

DDT has been tested against the C compiler xlc version 10.0 and Fortran/Fortran 90 version 12.1 – on both Linux and AIX. Note that xlC (C++) is not fully supported on AIX.

To view Fortran assumed size arrays in DDT you must first right click on the variable, select *Edit Type...*, and enter the type of the variable with its bounds (e.g. `integer arr(5)`).

MAP only supports xlc and xlf on Linux.

C.6 Intel Compilers

DDT has been tested with versions 10, 11 and 12.

If you do not see stack traces for allocations in the *View Pointer Details* window try re-compiling your program with the `-fno-omit-frame-pointer` argument (this enables frame pointers).

Some optimizations performed when `-ax` options are specified to IFC/ICC can result in programs which cannot be debugged. This is due to the reuse by the compiler of the frame-pointer, which makes DDT unable to obtain a stack trace.

© 2015 Allinea Software Ltd.

The Intel compiler doesn't always provide enough information to correctly determine the bounds of some Fortran arrays when they are passed as parameters, in particular the lower-bound of assumed-shape arrays.

The Intel OpenMP compiler will always optimise parallel regions, regardless of any `-O0` settings. This means that your code may jump around unexpectedly while stepping inside such regions, and that any variables which may have been optimised out by the compiler may be shown with nonsense values. There have also been problems reported in viewing thread-private data structures and arrays. If these affect you, please contact support@allinea.com .

Files with a `.F` or `.F90` extension are automatically preprocessed by the Intel compiler. This can also be turned on with the `-fpp` command-line option. Unfortunately, the Intel compiler does not include the correct location of the source file in the executable produced when preprocessing is used. If your Fortran file does not make use of macros and doesn't need preprocessing, you can simply rename its extension to `.f` or `.f90` and/or remove the `-fpp` flag from the compile line instead. Alternatively, you can help DDT discover the source file by right clicking in the *Project Files* window and then selecting *Add/view source directory* and adding the correct directory.

Some versions of the compiler emit incorrect debug information for OpenMP programs which may cause some OpenMP variables to show as `<not allocated>`.

By default Fortran `PARAMETERS` are not included in the debug information output by the Intel compiler. You can force them to be included by passing the `-debug-parameters all` option to the compiler.

Known Issue: If compiling static binaries (for example on a Cray XT/XE machine) then linking in the DDT memory debugging library is not straight forward for F90 applications. You will need to manually re-run the last `ld` command (as seen with `ifort -v`) to include `-L{ddt-path}/lib/64-ldmalloc` in two locations – both immediately prior to where `-lc` is located, and also include the `-zmuldefs` option at the start of the `ld` line.

Pretty printing of STL types is not supported for the Intel 10 compiler.

Pretty printing of STL types for the Intel 11 and 12 compiler is almost complete – STL sets, maps and multi-maps cannot be fully explored – only the total number of items is displayed; other data types are unaffected.

To disable pretty printing set the environment variable `DDT_DISABLE_PRETTY_PRINTING` to 1 before starting DDT. This will enable – in the case of, for example, the incomplete `std::set` implementations – you to manually inspect the variable.

C.7 Pathscale EKO compilers

Not supported by MAP.

Known issues: The default Fortran compiler options may not generate enough information for DDT to show where memory was allocated from – *View Pointer Details* will not show which line of source code memory was allocated from. To enable this, please compile and link with the following flags:

```
-Wl,--export-dynamic -TENV:frame_pointer=ON -funwind-tables
```

For C programs, simply compiling with `-g` is sufficient.

When using the Fortran compiler, you may have to place breakpoints in `myfile.i` instead of `myfile.f90` or `myfile.F90`. We are currently investigating this; please let us know if it applies to your code.

Procedure names in modules often have extra information appended to them. This does not otherwise affect the operation of DDT with the Pathscale compiler.

The Pathscale 3.1 OpenMP library has an issue which makes it incompatible with programs that call the fork system call on some machines.

Some versions of the Pathscale compiler (e.g. 3.1) do not emit complete DWARF debugging information for `typedef`'ed structures. These may show up in DDT with a `void` type instead of the expected type.

Multi-dimensional allocatable arrays can also be given incorrect dimension upper/lower bounds – this has only been reproduced for large arrays, small arrays seem to be unaffected. This has been observed with version 3.2 of the compiler, newer/older versions may also exhibit the same issue.

C.8 Portland Group Compilers

MAP has been tested with version 13.5 of the PGI compilers. Older versions are **not** supported as they do not allow line level profiling.

DDT has been tested with Portland Tools 9 onwards.

Known issues: Included files in Fortran 90 generate incorrect debug information with respect to file and line information. The information gives line numbers which refer to line numbers from the *included* file but give the *including* file as the file.

The PGI compiler may emit incorrect line number information for templated C++ functions or omit it entirely. This may cause DDT to show your program on a different line to the one expected, and also mean that breakpoints may not function as expected.

The PGI compiler does not emit the correct debugging tags for proper support of inheritance in C++, which prevents viewing of base class members.

When using memory debugging with statically linked PGI executables (`-Bstatic`) because of the in-built ordering of library linkage for F77/F90, you will need to add a `localrc` file to your PGI installation which defines the correct linkage when using DDT and (static) memory debugging. To your {`pgi-path`}`/bin/localrc` append the following:

```
switch -Bstaticddt is
 help(Link for DDT memory debugging with static binding)
 helpgroup(linker)
 append(LDARGS=--eh-frame-hdr –z muldefs)
 append(LDARGS=-Bstatic)
 append(LDARGS=-L{DDT-Install-Path}/lib/64)
 set(CRTL=$if(-Bstaticddt,-ldmallocthcxx -lc -lns$(PREFIX)c
 -l$(PREFIX)c, -lc -lns$(PREFIX)c -l$(PREFIX)c))
    set(LC=$if(-Bstaticddt,-ldmallocthcxx -lgcc -lgcc_eh -lc -lgcc
 -lgcc_eh -lc, -lgcc -lc -lgcc));
```

`pgf90 -help` will now list `-Bstaticddt` as a compilation flag. You should now use that flag for memory debugging with static linking.

This does not affect the default method of using PGI and memory debugging, which is to use dynamic libraries.

Note that some versions of `ld` (notably in SLES 9 and 10) silently ignore the `--eh-frame-hdr` argument in the above configuration, and a full stack for F90 allocated memory will not be shown in DDT.

© 2015 Allinea Software Ltd.

You can work around this limitation by replacing the system `ld`, or by including a more recent `ld` earlier in your path. This does not affect memory debugging in C/C++.

When you pass an array splice as an argument to a subroutine that has an assumed shape array argument, the offset of the array splice is currently ignored by DDT. Please contact support@allinea.com if this affects you.

DDT may show extra symbols for pointers to arrays and some other types. For example if your program uses the variable `ialloc2d` then the symbol `ialloc2d$sd` may also be displayed. The extra symbols are added by the compiler and may be ignored.

The Portland compiler also wraps F90 allocations in a compiler-handled allocation area, rather than directly using the systems memory allocation libraries directly for each allocate statement. This means that bounds protection (Guard Pages) cannot function correctly with this compiler.

For information concerning the Portland Accelerator model and debugging this with DDT, please see the 14 DDT: CUDA GPU Debugging of this userguide.

© 2015 Allinea Software Ltd.

D Platform Notes and Known Issues

This page notes any particular issues affecting platforms. If a supported machine is not listed on this page, it is because there is no known issue.

D.1 CRAY

For 'Native' SLURM mode systems GDB 7.6.2 must be selected as debugger (see Section 24.5.1 System) or the job might not start properly.

MAP users on Cray need to read 17.1.1 Debugging Symbols and 17.1.4 Static Linking on Cray X-Series Systems.

Note that the default mode for compilers on this platform is to link statically. Section C.8 Portland Group Compilers describes how to ensure that DDT's memory debugging capabilities will work with the PGI compilers in this mode.

Message queue debugging is not provided by the XT/XE/XK environment.

Cray XK6/7 GPU debugging requires the CUDA Toolkit 5 or above to be installed.

Cray XK6/7 GPU debugging requires a working `TMPDIR` to be available, if /tmp is not available. It is important that this directory is not a shared filesystem such as NFS or Lustre. To set `TMPDIR` for the compute nodes only use the `DDT_BACKEND_TMPDIR` environment variable instead (DDT will automatically propagate this to the compute nodes).

Running single process scalar codes (ie. *non*-MPI/SHMEM/UPC applications) on the compute nodes requires an extra step – as these are required to be executed by `aprun` but `aprun` will not execute these via the ordinary debug-supporting protocols.

The preferred and simple workaround is to use the `.qtf` templates (eg. `cray-slurm.qtf` or `cray-pbs.qtf`) which handle this automatically by (for non-MPI codes) ensuring that an alternative protocol is followed. To use these `qtf` files, select *File* → *Options* (*DDT* → *Preferences* on Mac OS X) , go to the *Job Submission* page and enable submission via the queue, and ensure that the *Also submit scalar jobs via the queue* setting is enabled. The change is to explicitly use aprun for non-MPI processes and this can be seen in the provided queue template files:

```
if [ "MPI_TAG" == "none" ]; then
   aprun -n 1 env AUTO_LAUNCH_TAG
else
   AUTO_LAUNCH_TAG
fi
```

Running a dynamically-linked single process non-MPI program that will run on a compute node (i.e. non-MPI CUDA or OpenACC code) will require an additional flag to the compiler: `-target=native` - this prevents the compiler linking in the MPI job launch routines that will otherwise interfere with debuggers on this platform. Alternatively, convert the program to an MPI one by adding `MPI_Init` and `MPI_Finalize` statements and run it as a one-process MPI job.

© 2015 Allinea Software Ltd.

D.2 GNU/Linux Systems

D.2.1 General

When using a 64-bit Linux please note that it is essential to use the 64-bit version of DDT/MAP on this platform. This applies regardless of whether the debugged program is 32-bit or 64-bit.

POSIX thread cancellation does not work when running under a debugger. This is because the 'signal info' associated with a signal is lost when the signal is intercepted and resent by the debugger, causing the cancellation request to be ignored by the receiving thread.
More generally the 'signal info' associated with a signal is not available when running under a debugger.

Some 64-bit GNU/Linux systems which have a bug in the GNU C library (specifically `libthread_db.so.1`) which can crash the debugger when debugging multi-threaded programs. Check with your Linux distribution for a fix. As a workaround you can try compiling your program as a statically linked executable using the `-static` compiler flag.

For the ARM architecture – breakpoints can be unreliable and will randomly be passed without stopping for some multicore processors (including the NVIDIA Tegra 2) unless a kernel option (fix) is built-in. The required kernel option is:

```
CONFIG_ARM_ERRATA_720789=y
```

This option is not present by default in many kernel builds.

D.2.2 SUSE Linux

There is a known issue with SUSE 11 which *may* cause you to experience a crash similar to this

```
Other: *** glibc detected *** /home/user/wave_c: free(): invalid
   pointer: 0x00007ffff7e02a80 ***
Other: ======= Backtrace: =========
Other: /lib64/libc.so.6[0x7fffeef81118]
Other: /lib64/libc.so.6(cfree+0x76)[0x7fffeef82c76]
Other: /lib64/libnss_nis.so.2(_nss_nis_getpwuid_r+0xe9)[0
   x7fffecd4f089]
Other: /lib64/libnss_compat.so.2[0x7fffed125ab8]
```

The implementation of `libnss_nis.so.2` attempts to resolve symbol names using its direct dependencies before using the global namespace. This causes the libc implementation of, for example, `free` to be linked instead of the intended `libdmalloc` implementation.

If you encounter this crash, then the only solution is to disable memory debugging and contact SUSE about the availability of a fix.

D.3 IBM AIX Systems

MAP is not supported on AIX.

The *Step Threads Together* and *Focus on Thread* features are unavailable on AIX due to a lack of operating system support.

When stepping through certain system library calls the program may run freely instead. This is presently known to occur with the `memset` library call, for example. This is due to a limitation of AIX.

A sample Loadleveller script, which starts debugging jobs on IBM AIX (POE) systems is included in the {*installation-directory*}/templates directory.

The stop on fork and stop on exec feature is not supported on this platform.

D.4 IBM Blue Gene/Q

MAP is not supported on Blue Gene.

DDT must be installed in a directory that is visible from the front end node(s), the service nodes and the I/O nodes.

Message queues are not currently supported on this platform.

D.4.1 Attaching

To attach to a running job:

1. Open the *Attach* window by clicking on the *Attach* button on the Welcome page.
2. DDT needs to know which login / batch node `runjob` is running on. Click the *Choose Hosts...* button to add the necessary login / batch node if not already present. You must be able to SSH into the login / batch node without a password.
3. Select the *Automatically-detected jobs* tab. Do not use the *List of processes* tab.
4. Optionally specify a subset of ranks to attach to in the *Attach to processes* box.
5. Click the *Attach to...* button.

The following caveats apply:

- Re-attaching to a job is not supported. You may only attach to a job once.
- No other tool must be attached, or have been attached, to the job.
- It is possible to attach to a subset of ranks. However, because re-attaching is not supported, it is not possible to subsequently change the subset.
- It may take a little time for a job to show up in the Attach window after you submit it. If a newly started job does not show up wait a while then click *Rescan nodes*.

D.5 Intel Xeon Phi

D.5.1 Requirements

	MPSS Minimum Version
DDT	2.1.4982-15
Offload Support (DDT)	2.1.6720-13
MAP	2.1.6720-19

Important: All Intel MPSS 3.1 and 3.2 releases at the time of writing suffer from a serious issue which prevents debugging offload programs. The recommend workaround from Intel is to copy the debug information for the system libraries to the Xeon Phi card:

```
[host]
$ scp -r /opt/mpss/3.1.2/sysroots/k1om-mpss-linux/lib64/.debug
   root@mic0:/lib64/
```

Please contact Allinea for up to date information.

D.5.2 Installation

To debug or profile programs running on Intel Xeon Phi cards you need to download and install the relevant combined host and Xeon Phi installation tarball for your host machine.

The Allinea Tools installation for the Xeon Phi card must be accessible from the Xeon Phi card itself, either using NFS (recommended) or a filesystem overlay (not recommended as it reduces the available memory).

See also the section E.9.7 MAP harmless error messages in Xeon Phi.

Note: Your existing licence may not support debugging on the Intel Xeon Phi card. If you have the coprocessor option in your licence please contact Allinea for a free upgrade.

D.5.3 Configuration

	DDT	MAP
Native Xeon Phi non-MPI Programs	remote	remote
Native Xeon Phi Intel MPI Programs	remote	remote
Native Xeon Phi Cray MPT Programs	GUI / offline	GUI / offline
Heterogeneous Intel MPI Programs	GUI / offline	GUI / offline
Heterogeneous Cray MPT Programs	GUI / offline	GUI / offline
Heterogeneous Programs (#pragma offload)	GUI / offline	GUI / offline

Native Xeon Phi non-MPI Programs

Debugging

Note: The DDT GUI can not run on the Xeon Phi card directly.

To debug a native Xeon Phi non-MPI program:

1. Start DDT on the host (using the host installation of DDT).
2. Click the *Remote Launch* drop-down on the *Welcome Page* and select *Configure...*
3. Enter the host name of the Xeon Phi card (e.g. `micdev-mic0`) in the Host Name box.
4. Select the path to the Xeon Phi installation of DDT in the *Installation Directory* box.
5. Click *Test Remote Launch* and ensure the settings are correct.
6. Click *Ok*.
7. Click *Run and Debug a Program* on the *Welcome Page*.
8. Select a native Xeon Phi program in the *Application* box in the Run window.
9. Click *Run*.

© 2015 Allinea Software Ltd.

Profiling

To profile a native Xeon Phi non-MPI program:

1. Start MAP on the host (using the host installation of MAP).
2. Click the *Remote Launch* drop-down on the *Welcome Page* and select *Configure...*
3. Enter the host name of the Xeon Phi card (e.g. `micdev-mic0`) in the *Host Name* box.
4. Select the path to the Xeon Phi installation of MAP in the *Installation Directory* box.
5. Click *Test Remote Launch* and ensure the settings are correct.
6. Click *Ok*.
7. Click *Profile a program* on the *Welcome Page*.
8. Select your native Xeon Phi MPI program in the *Application* box in the *Run* window.
9. Click *Run*.

Native Xeon Phi Intel MPI Programs

Debugging

Note: The DDT GUI can not run on the Xeon Phi card directly.

To debug a native Xeon Phi Intel MPI program:

1. Start DDT on the host (using the host installation of DDT).
2. Click the *Remote Launch* drop-down on the *Welcome Page* and select *Configure...*
3. Enter the host name of the Xeon Phi card (e.g. `micdev-mic0`) in the Host Name box.
4. Select the path to the Xeon Phi installation of DDT in the *Installation Directory* box.
5. Click *Test Remote Launch* and ensure the settings are correct.
6. Click *Ok*.
7. Click *Run and Debug a Program* on the *Welcome Page*.
8. Select a native Xeon Phi Intel MPI program in the Application box in the Run window.
    - DDT should have detected 'Intel MPI (MPMD)' as the MPI implementation in *File* → *Options* (*DDT* → *Preferences* on Mac OS X) → *System*.
9. Click *Run*.

Profiling

To profile a native Xeon Phi Intel MPI program:

1. Ensure the Intel Compilers and MPI are in your path.
2. Start MAP on the host (using the host installation of MAP).
3. Click the *Remote Launch* drop-down on the *Welcome Page* and select *Configure...*
4. Enter the host name of the Xeon Phi card (e.g. `micdev-mic0`) in the Host Name box.
5. Select the path to the Xeon Phi installation of MAP in the *Installation Directory* box.
6. Click *Test Remote Launch* and ensure the settings are correct.
7. Click *Ok*.

© 2015 Allinea Software Ltd.

8. Click *Profile a program* on the *Welcome Page*.
9. Select your native Xeon Phi Intel MPI program in the Application box in the Run window.
    - MAP should have detected 'Intel MPI (MPMD)' as the MPI implementation in *File* → *Options* (*MAP* → *Preferences* on Mac OS X) → *System*.
10. Click *Run*.

Native Xeon Phi Cray MPT Programs

Debugging

Note: The DDT GUI can not run on the Xeon Phi card directly.

To debug a native Xeon Phi Cray MPT program:

1. Start DDT on the login node or host (using the host installation of DDT).
2. Open the *Options* window: *File* → *Options* (*DDT* → *Preferences* on Mac OS X) .
3. Select *Intel MPI (MPMD)* as the *MPI Implementation* on the *System* page.
4. Check the *Heterogeneous system support* check box on the *System* page.
5. Click *Run and Debug a Program* on the *Welcome Page*.
6. Select a native Xeon Phi Cray MPT program in the Application box in the Run window.
    - DDT should have detected 'Cray MPT' as the MPI implementation in *File* → *Options* (*DDT* → *Preferences* on Mac OS X) → *System*.
7. Add the `-k` argument to the *`aprun` Arguments* box.
8. Click *Run*.

Profiling

To profile a native Xeon Phi Intel MPI program:

1. Start MAP on the login node or host (using the host installation of DDT).
2. Open the *Options* window: *File* → *Options* (*DDT* → *Preferences* on Mac OS X) .
3. Select *Intel MPI (MPMD)* as the *MPI Implementation* on the *System* page.
4. Check the *Heterogeneous system support* check box on the *System* page.
5. Click *Profile a program* on the *Welcome Page*.
6. Select your native Xeon Phi Cray MPT program in the Application box in the Run window.
    - MAP should have detected 'Cray MPT' as the MPI implementation in *File* → *Options* (*MAP* → *Preferences* on Mac OS X) → *System*.
7. Add the `-k` argument to the *`aprun` Arguments* box.
8. Click *Run*.

Heterogeneous (host + Xeon Phi) Intel MPI Programs

Debugging

To debug a heterogeneous (host + Xeon Phi) Intel MPI program:

1. Start DDT on the host (using the host installation of DDT).

2. Open the *Options* window: *File* → *Options* (*DDT* → *Preferences* on Mac OS X) .
3. Select *Intel MPI (MPMD)* as the *MPI Implementation* on the *System* page.
4. Check the *Heterogeneous system support* check box on the *System* page.
5. Click *Ok*.
6. Click *Run and Debug a Program* in the *Welcome Page*.
7. Select the path to the *host* executable in the *Application box* in the *Run window*.
8. Enter an MPMD style `mpiexec`command line in the *mpiexec Arguments* box, e.g.

```
-np 8 -host micdev /home/user/examples/hello-host : -np 32 -
   host micdev-mic0 /home/user/examples/hello-mic
```

9. Set *Number of processes* to be the total number of processes launched on both the host and Xeon Phi (e.g. 40 for the above *mpiexec Arguments* line).
10. Add `I_MPI_MIC=enable` to the *Environment Variables* box.
11. Click *Run*. You may need to wait a minute for the Xeon Phi processes to connect.

Profiling

To profile a native Xeon Phi Intel MPI program:

1. Open the *Options* window: *File* → *Options* (*MAP* → *Preferences* on Mac OS X) .
2. Select *Intel MPI (MPMD)* as the *MPI Implementation* on the *System* page.
3. Check the *Heterogeneous system support* check box on the *System* page.
4. Click *Ok*.
5. Click *Run and Debug a Program* in the *Welcome Page*.
6. Select the path to the *host* executable in the *Application box* in the *Run window*.
7. Enter an MPMD style `mpiexec`command line in the *mpiexec Arguments* box, e.g.

```
-np 8 -host micdev /home/user/examples/wave-host : -np 32 -
   host micdev-mic0 /home/user/examples/wave-xeon-phi
```

8. Set *Number of processes* to be the total number of processes launched on both the host and Xeon Phi (e.g. 40 for the above *mpiexec Arguments* line).
9. Add `I_MPI_MIC=enable` to the *Environment Variables* box.
10. Click *Run*. You may need to wait a minute for the Xeon Phi processes to connect.

Heterogeneous Programs (#pragma offload)

Intel recommend setting the following environment variables before debugging offload programs:

```
COI_SEP_DISABLE=FALSE
AMPLXE_COI_DEBUG_SUPPORT=TRUE
MYO_WATCHDOG_MONITOR=-1
```

The `OFFLOAD_MAIN` environment must be unset, or set to `on_offload` or `on_offload_all` when debugging offload programs in DDT. If `OFFLOAD_MAIN` is set to `on_start` then DDT will not attach to the offloading host processes.

© 2015 Allinea Software Ltd.

Memory debugging is not supported for programs that use `#pragma offload`.

Debugging

When debugging offloaded code (i.e. code offloaded to the Xeon phi using `#pragma offload`) DDT can automatically attach to the offload process running on the Xeon Phi Card.

To debug a heterogeneous program that uses `#pragma offload`:

1. Start DDT on the host (using the host installation of DDT).
2. Open the *Options* window: *File* → *Options* (*DDT* → *Preferences* on Mac OS X) .
3. Select *Intel MPI (MPMD)* as the *MPI Implementation on the System* page.
4. Check the *Heterogeneous system support* check box on the *System* page.
5. Click *Ok*.
6. *Ensure Control* → *Default Breakpoints* → *Stop on Xeon Phi offload* is checked.
7. Click *Run and Debug a Program* on the *Welcome Page*.
8. Select a heterogeneous program that uses `#pragma offload` in the *Application* box in the *Run* window.
9. Click *Run*.

Profiling

MAP does not support profiling of offloaded code (i.e. code offloaded to the Xeon phi using `#pragma offload`). The host portion of the program may be profiled as normal.

© 2015 Allinea Software Ltd.

E General Troubleshooting and Known Issues

If you have any problems with DDT or MAP, please take a look at the topics in this section – you might just find the answer you're looking for. Equally, it's worth checking the support pages of http://www.allinea.com and making sure you have the latest version.

E.1 General Troubleshooting

E.1.1 Problems Starting the GUI

If the GUI is unable to start this is usually due to one of three reasons:

- Cannot connect to an X server. If you are running on a remote machine, make sure that your `DISPLAY` variable is set appropriately and that you can run simple X applications such as `xterm` from the same command-line.
- The licence file is invalid – in this case the software will issue an error message. You should verify that you have a licence file for the correct product in the licence directory and check that the date inside it is still valid. If DDT/MAP still refuses to start, please contact Allinea.
- You are using a licence server, but DDT/MAP cannot connect to it. See the section 25 The Licence Server for more information on troubleshooting these problems.

E.1.2 Problems Reading this document

If when pressing F1 a blank screen appears instead of this document, there may be corrupt files that are preventing the documentation system (Qt Assistant) from starting. You can resolve this by removing the stale files, which are found in `$HOME/.local/share/data/Allinea`.

E.2 Starting a Program

E.2.1 Problems Starting Scalar Programs

There are a number of possible sources for problems. The most common is – for users with a multi-process licence – that the *Run Without MPI Support* check box has not been checked. If the software reports a problem with MPI and you know your program is not using MPI, then this is usually the cause. If you HAVE checked this box and the software still mentions MPI then we would very much like to hear from you!

Other potential problems are:

- A previous DDT/MAP session is still running, or has not released resources required for the new session. Usually this can be resolved by killing stale processes. The most obvious symptom of this is a delay of approximately 60 seconds and a message stating that not all processes connected. You may also see, in the terminal, a `QServerSocket` message
- The target program does not exist or is not executable
- DDT/MAP's backend daemon – `ddt-debugger` – is missing from the `bin` directory – in this case you should check your installation, and contact Allinea for further assistance.

© 2015 Allinea Software Ltd.

E.2.2 Problems Starting Multi-Process Programs

If you encounter problems whilst starting an MPI program, the first step is to establish that it is possible to run a single-process (non-MPI) program such as a trivial "Hello, World!" - and resolve such issues that may arise. After this, attempt to run a multi-process job – and the symptoms will often allow a reasonable diagnosis to be made.

In the first instance, verify that MPI is installed correctly by running a job outside of DDT/MAP, such as the example in the examples directory.

```
mpirun -np 8 ./a.out
```

Verify that `mpirun`is in the `PATH`, or the environment variable `DDT_MPIRUN/MAP_MPIRUN` is set to the full pathname of `mpirun`.

If the progress bar does not report that at least process 0 has connected, then the remote `ddt-debugger` daemons cannot be started or cannot connect to the GUI.

Sometimes problems are caused by environment variables not propagating to the remote nodes whilst starting a job. To a large extent, the solution to these problems depend on the MPI implementation that is being used. In the simplest case, for rsh based systems such as a default MPICH 1 installation, correct configuration can be verified by rsh-ing to a node and examining the environment. It is worthwhile rsh-ing with the env command to the node as this will not see any environment variables set inside the .profile command. For example if your nodes use a `.profile` instead of a `.bashrc` for each user then you may well see a different output when running `rsh node env` than when you `run rsh` node and then run `env` inside the new shell.

If only one, or very few, processes connect, it may be because you have not chosen the correct MPI implementation. Please examine the list and look carefully at the options. Should no other suitable MPI be found, please contact Allinea for advice.

If a large number of processes are reported by the status bar to have connected, then it is possible that some have failed to start due to resource exhaustion, timing out, or, unusually, an unexplained crash. You should verify again that MPI is still working, as some MPI distributions do not release all semaphore resources correctly (for example MPICH 1 on Redhat with SMP support built in).

To check for time-out problems, set the `MAP/DDT_NO_TIMEOUT` environment variable to 1 before launching the GUI and see if further progress is made. This is not a solution, but aids the diagnosis. If all processes now start, please contact Allinea for further long-term advice.

E.2.3 No Shared Home Directory

If your home directory is not accessible by all the nodes in your cluster then your jobs may fail to start. To resolve the problem open the file `~/.allinea/config.system` in a text editor. Change the `shared directory` option in the `[startup]` section so it points to a directory that is available and shared by all the nodes. If no such directory exists, change the `use session cookies` option to `no` instead.

E.2.4 DDT/MAP says it can't find your hosts or the executable

This can happen when attempting to attach to a process running on other machines. Ensure that the host name(s) that DDT/MAP complains about are reachable using `ping`. If DDT/MAP fails to find the executable, ensure that it is available in the same directory on every machine. See section 24.4 Connecting to remote programs (remote-exec) for more information on configuring access to remote machines.

E.2.5 The progress bar doesn't move and DDT/MAP 'times out'

It's possible that the program `ddt-debugger` hasn't been started by `mpirun`or has aborted. You can log onto your nodes and confirm this by looking at the process list **before** clicking *Ok* when DDT/MAP times out. Ensure `ddt-debugger` has all the libraries it needs and that it can run successfully on the nodes using `mpirun`.

Alternatively, there may be one or more processes (`ddt-debugger`, `mpirun`, `rsh`) which could not be terminated. This can happen if DDT/MAP is killed during its startup or due to MPI implementation issues. You will have to kill the processes manually, using `ps x` to get the process ids and then `kill` or `kill -9` to terminate them.

This issue can also arise for `mpich-p4mpd`, and the solution is explained in Appendix B MPI Distribution Notes and Known Issues.

If your intended `mpirun`command is not in your `PATH`, you may either add it to your `PATH` or set the environment variable `DDT_MPIRUN/MAP_MPIRUN` to contain the full pathname of the correct `mpirun`.

If your home directory is not accessible by all the nodes in your cluster then your jobs may fail to start in this fashion. See section E.2.3 No Shared Home Directory.

E.2.6 The progress bar gets close to half the processes connecting and then stops and DDT/MAP 'times out'

This is likely to be caused by a dual-processor configured MPI distribution. Make sure you have selected *smp-mpich* or *scyld* as your MPI implementation in the Options window. If this doesn't help, see Appendix B MPI Distribution Notes and Known Issues for a workaround and email support@allinea.com for further assistance.

E.3 Attaching

E.3.1 The system does not allow attaching to processes (Ubuntu)

The Ubuntu ptrace scope control feature does not allow a process to attach to other processes it did not launch directly (see http://wiki.ubuntu.com/Security/Features#ptrace for details).

To disable this feature until the next reboot run the following command:

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

To disable it permanently add this line to `/etc/sysctl.conf`:

```
kernel.yama.ptrace_scope = 0
```

(this will take effect after the next reboot)

E.3.2 The system does not allow attaching to processes (Fedora, Red Hat)

The `deny_ptrace` boolean in SELinux (used by Fedora and Red Hat) does not allow a process to attach to other processes it did not launch directly (see http://fedoraproject.org/wiki/Features/SELinuxDenyPtrace for details).

To disable this feature until the next reboot run the following command:

```
setsebool deny_ptrace 0
```

To disable it permanently run this command:

```
setsebool -P deny_ptrace 0
```

E.3.3 Running processes don't show up in the attach window

This is usually a problem with either your `remote-exec` script or your node list file. First check that the entry in your node list file corresponds with either localhost (if you're running on your local machine) or with the output of `hostname` on the desired machine.

Secondly try running `/path/to/allinea-tools/libexec/remote-exec` manually ie. `/path/to/allinea-tools/libexec/remote-exec<hostname>ls` and check the output of this. If this fails then there is a problem with your `remote-exec` script. If `rsh` is still being used in your script check that you can rsh to the desired machine. Otherwise check that you can attach to your machine in the way specified in the `remote-exec` script. (See also 24.4 Connecting to remote programs (remote-exec)) If you still experience problems with your script then contact Allinea for assistance.

E.4 Source Viewer

E.4.1 No variables or line number information

You should compile your programs with debug information included, this flag is usually `-g`.

E.4.2 Source code does not appear when you start DDT / MAP

If you cannot see any text at all, perhaps the default selected font is not installed on your system. Go to *File* → *Options* (*DDT* → *Preferences* on Mac OS X) and choose a fixed width font such as *Courier* and you should now be able to see the code.

If you see a screen of text telling you that DDT / MAP could not find your source files, follow the instructions given. If you still cannot see your source code, check that the code is available on the same machine as you are running the software on, and that the correct file and directory permissions are set. If some files are missing, and others found, try adding source directories and rescanning for further instruction.

If the problem persists, contact support@allinea.com .

E.5 Input/Output

E.5.1 Output to stderr is not displayed

DDT / MAP automatically captures anything written to `stdout` / `stderr` and displays it. Some shells (such as `csh`) do not support this feature in which case you may see your `stderr` mixed with `stdout`, or you may not see it at all. In any case we strongly recommend writing program output to files instead, since the MPI specification does not cover `stdout` / `stderr` behaviour.

© 2015 Allinea Software Ltd.

E.6 Controlling a Program

E.6.1 Program jumps forwards and backwards when stepping through it

If you have compiled with any sort of optimisations, the compiler will shuffle your programs instructions into a more efficient order. This is what you are seeing. We always recommend compiling with `-O0` when debugging, which disables this behaviour and other optimisations.

If you are using the Intel OpenMP compiler, then the compiler will generate code that appears to jump in and out of the parallel blocks regardless of your `-O0` setting. Stepping inside parallel blocks is therefore not recommended for the faint-hearted!

E.6.2 DDT sometimes stop responding when using the Step Threads Together option

DDT may stop responding if a thread exits when the *Step Threads Together* option is enabled. This is most likely to occur on Linux platforms using NPTL threads. This might happen if you tried to *Play to here* to a line that was never reached – in which case your program ran all the way to the end and then exited.

A workaround is to set a breakpoint at the last statement executed by the thread and turn off *Step Threads Together* when the thread stops at the breakpoint. If this problem affects you please contact support@allinea.com .

E.7 Evaluating Variables

E.7.1 Some variables cannot be viewed when the program is at the start of a function

Some compilers produce faulty debug information, forcing DDT to enter a function during the *prologue* or the variable may not yet be in scope. In this region, which appears to be the first line of the function, some variables have not been initialised yet. To view all the variables with their correct values, it may be necessary to play or step to the next line of the function.

E.7.2 Incorrect values printed for Fortran array

Pointers to non-contiguous array blocks (allocatable arrays using strides) are not supported. If this issue affects you, please email support@allinea.com for a workaround or fix. There are also many compiler limitations that can cause this. See Appendix C for details.

E.7.3 Evaluating an array of derived types, containing multiple-dimension arrays

The *Locals*, *Current Line* and *Evaluate* views may not show the contents of these multi-dimensional arrays inside an array of derived types. However, you can view the contents of the array by clicking on its name and dragging it into the evaluate window as an item on its own, or by using the MDA

© 2015 Allinea Software Ltd.

E.7.4 C++ STL types are not pretty printed

The pretty printers provided with Allinea DDT are compatible with GNU compilers, and Intel C++ version 12 and above.

E.7.5 The Fortran Module Browser is missing

Not all platforms support every feature that DDT has and so they are disabled by removing the window/tab by from DDT's interface. The Fortran modules browser is not supported on AIX.

E.8 Memory Debugging

E.8.1 The View Pointer Details window says a pointer is valid but doesn't show you which line of code it was allocated on

The Pathscale compilers have known issues that can cause this – please see the compiler notes in section C of this appendix for more details.

The Intel compiler may need the `-fp` argument to allow you to see stack traces on some machines.

If this happens with another compiler, please contact support@allinea.com with the vendor and version number of your compiler.

E.8.2 `mprotect fails` error when using memory debugging with guard pages

This can happen if your program makes more than 32768 allocations – a limit in the kernel prevents DDT from allocating more protected regions than this. You can set this limit manually by logging in as root and executing `echo 1048576 >/proc/sys/vm/max_map_count`, or another limit of your choice.

E.8.3 Allocations made before or during `MPI_Init` show up in *Current Memory Usage* but have no associated stack back trace

Memory allocations that are made before or during `MPI_Init` appear in Current Memory Usage along with any allocations made afterwards. However the call stack at the time of the allocation is not recorded for these allocations and will not show up in the Current Memory Usage window.

E.8.4 Deadlock when calling `printf` or `malloc` from a signal handler

The memory allocation library calls (e.g. `malloc`) provided by the memory debugging library are not async-signal-safe unlike the implementations in recent versions of the GNU C library.

POSIX does not require `malloc` to be async-signal-safe but some programs may expect this behaviour. For example a program that calls `printf` from a signal handler may deadlock when memory debugging is enabled in DDT since the C library implementation of `printf` may call `malloc`.

The web page below has a table of the functions that may be safely called from an asynchronous signal handler:

http://www.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_04.html#tag_02_04_03/

© 2015 Allinea Software Ltd.

E.8.5 Program runs more slowly with Memory Debugging enabled

The Memory Debugging library performs more checks than the normal runtime's memory allocation routines – that's what makes it a debugging library! However those checks also makes it slower. If your program is running too slow when Memory Debugging is enabled there are a number of options you can change to speed it up.

Firstly try reducing the *Heap Debugging* option to a lower setting (e.g. if it is currently on *High*, try changing it to *Medium* or *Low*).

You can increase the heap check interval from the default of 100 to a higher value. The heap check interval controls how many allocations may occur between full checks of the heap (which may take some time). A higher setting (1000 or above) is recommended if your program allocates and deallocates memory very frequently (e.g. inside a computation loop).

You can disable the *Store backtraces for memory allocations* option, at the expense of losing backtraces in the *View Pointer Details* and *Current Memory Usage* windows.

E.9 MAP specific issues

E.9.1 My compiler is inlining functions

Yes, they do that. Unfortunately their abilities to include sufficient information to reconstruct the original call tree vary between vendors. We've found that the following flags work best:

- Intel: -g -O3 -fno-inline-functions
- PGI: -g -Mprof=func -O3
- GNU: -g -O3 -fno-inline

Be aware that some compilers may still inline functions even when explicitly asked not to.

There is typically some small performance penalty for disabling function inlining or enabling profiling information; depending on your code you may see around an 8% slowdown with the PGI compiler's -Mprof=func option, for example.

Alternatively, you can let the compiler inline the functions and just compile with -g -O3. Or -g -O5 or whatever your preferred performance flags are. MAP will work just fine, but you will often see time inside an inlined function being attributed to its parent in the Stacks and Project Files views. The Source Code view should be largely unaffected.

E.9.2 MPI Wrapper Libraries

Unlike DDT, MAP needs to wrap MPI calls in a custom shared library. We build one, just for your system, each time you run MAP. Sometimes it won't work. If it doesn't, please tell us. It should work on every system we've ever seen, first time, every time. In the meantime, you can also try setting MPICC directly:

```
$ MPICC=my-mpicc-command bin/map -n 16 ./wave_c
```

© 2015 Allinea Software Ltd.

E.9.3 I'm not getting enough samples

By default MAP samples every 20ms, but if you get warnings about too few samples on a fast run, or want more detail in the results, you can change that. To increase the frequency to every 10ms set environment variable `MAP_INTERVAL=10`.

E.9.4 I just see main (external code) and nothing else

This can happen if you compile without -g. It can also happen if you move the executable out of the directory it was compiled in. Tell us if it's happened to you; in the meantime check your compile line includes -g and try right-clicking on the Project Files panel in MAP and choosing "Add Source Directory..."

E.9.5 MAP is reporting time spent in a function definition

Any overheads involved in setting up a function call (pushing arguments to the stack etc) are usually assigned to the function definition. Some compilers may assign them to the opening brace '{' and closing brace '}' instead. If this function has been inlined, the situation becomes further complicated and any setup time (e.g. allocating space for arrays) is often assigned to the definition line of the enclosing function.

We're looking for ways to unravel this and present a more intuitive picture; any ideas or suggestions are much appreciated!

E.9.6 MAP is not correctly identifying vectorized instructions

The instructions MAP identifies as vectorized (packed) are enumerated below. We also identify the AVX-2 variants of these instructions (with a "v" prefix). Contact support@allinea.com if you believe your code contains vectorized instructions that have not been listed and are not being identified in the *CPU floating-point/integer vector* metrics.

Packed floating-point instructions: `addpd addps addsubpd addsubps andnpd andnps andpd andps divpd divps dppd dpps haddpd haddps hsubpd hsubps maxpd maxps minpd minps mulpd mulps rcpps rsqrtps sqrtpd sqrtps subpd subps`

Packed integer instructions: `mpsadbw pabsb pabsd pabsw paddb paddd paddq paddsb paddsw paddusb paddusw paddw palignr pavgb pavgw phaddd phaddsw phaddw phminposuw phsubd phsubsw phsubw pmaddubsw pmaddwd pmaxsb pmaxsd pmaxsw pmaxub pmaxud pmaxuw pminsb pminsd pminsw pminub pminud pminuw pmuldq pmulhrsw pmulhuw pmulhw pmulld pmullw pmuludq pshufb pshufw psignb psignd psignw pslld psllq psllw psrad psraw psrld psrlq psrlw psubb psubd psubq psubsb psubsw psubusb psubusw psubw`

E.9.7 MAP harmless error messages in Xeon Phi

When running MAP on a Xeon Phi host, where the MAP installation has been configured for D.5 Intel Xeon Phi heterogeneous support, but your MPI program was compiled without MIC options, you may see harmless 'ERROR' messages similar to the following:

Other: ERROR: ld.so: object '/home/user/.allinea/wrapper/libmap-sampler-pmpi-mic3-mic-115427.so' from `LD_PRELOAD` cannot be preloaded: ignored.

These may be safely ignored.

E.9.8 MAP takes an extremely long time to gather and analyze my OpenBLAS-linked application

OpenBLAS versions 0.2.8 and earlier incorrectly stripped symbols from the .symtab section of the library, causing binary analysis tools such as Allinea MAP and objdump to see invalid function lengths and addresses.

This causes Allinea MAP to take an extremely long time disassembling and analyzing apparently overlapping functions containing millions of instructions.

A fix for this was accepted into the OpenBLAS codebase on October 8th 2013 and versions 0.2.9 and above should not be affected.

To work around this problem without updating OpenBLAS, simply run “strip libopenblas*.so” - this removes the incomplete .symtab section without affecting the operation or linkage of the library.

E.10 Obtaining Support

If this guide hasn't helped you, then the most effective way to get support is to email us with a detailed report. If possible, you should obtain a log file for the problem and email this to support@allinea.com .

You can generate a log file by checkingthe *Help* → *Logging* → *Automatic* menu item.

Then simply reproduce the problem using as few processors and commands as possible and close the program as usual. On some systems this file might be quite large; if this is the case, please compress it using a program such as `gzip` or `bzip2` before attaching it to your email.

If your problem can only be replicated on large process counts, then please do not use the *Help* → *Logging* → *Debug* menu item as this will generate very large log files – it will usually be sufficient to just use the *Help* → *Logging* → *Standard* option.

If you are connecting to a remote system, then the log file is generated on the remote host and copied back to the client when the connection is closed. The copy will not happen if the target application crashes or the network connection is lost. In these cases, the remote copy of the log file can be found in the `tmp` subdirectory of the Allinea configuration directory for the remote user account (which is `~/.allinea`, unless overridden by the `DDTCONFIGDIR` environment variable).

© 2015 Allinea Software Ltd.

F Queue Template Script Syntax

F.1 Queue Template Tags

Each of the tags that will be replaced is listed in the following table – and an example of the text that will be generated when DDT/MAP submits your job is given for each.

Note: It is often sufficient to simply use `AUTO_LAUNCH_TAG`*. See section 24.3.1 The Template Script for an example.*

Tag	Description	After Submission Example
AUTO_LAUNCH_TAG	This tag expands to the entire replacement for your 'mpirun' command line.	ddt-mpirun -np 4 myexample.bin
DDTPATH_TAG	The path to the DDT/MAP installation	/opt/allinea
WORKING_DIRECTORY_TAG	The working directory DDT/MAP was launched in	/users/ned
NUM_PROCS_TAG	Total number of processes	16
NUM_PROCS_PLUS_ONE_TAG	Total number of processes + 1	17
NUM_NODES_TAG	Number of compute nodes	8
NUM_NODES_PLUS_ONE_TAG	Number of compute nodes + 1	9
PROCS_PER_NODE_TAG	Processes per node	2
PROCS_PER_NODE_PLUS_ONE_TAG	Processes per node + 1	3
NUM_THREADS_TAG	Number of OpenMP threads per node (empty if OpenMP if “off”)	4
OMP_NUM_THREADS_TAG	Number of OpenMP threads per node (empty if OpenMP is “off”)	4
MPIRUN_TAG	mpirun binary (can vary with MPI implementation)	/usr/bin/mpirun
AUTO_MPI_ARGUMENTS_TAG	Required command line flags for mpirun (can vary with MPI implementation)	-np 4
EXTRA_MPI_ARGUMENTS_TAG	Additional mpirun arguments specified in the Run window	-partition DEBUG
PROGRAM_TAG	Target path and filename	/users/ned/a.out
PROGRAM_ARGUMENTS_TAG	Arguments to target program	-myarg myval
INPUT_FILE_TAG	The stdin file specified in the Run window	/users/ned/input.dat

Additionally, any environment variables in the GUI environment ending in _TAG are replaced throughout the script by the value of those variables.

© 2015 Allinea Software Ltd.

F.2 Defining New Tags

As well as the pre-defined tags listed in the table above you can also define new tags in your template script whose values can be specified in the GUI.

Tag definitions have the following format:

```
EXAMPLE_TAG: { key1=value1, key2=value2, ... }
```

Where `key1`, `key2`, … are attribute names and `value1`, `value2`, … are the corresponding values.

The tag will be replaced wherever it occurs with the value specified in the GUI, for example:

```
#PBS –option EXAMPLE_TAG
```

The following attributes are supported:

Attribute	Purpose	Example
`type`	`text:` general text input `select:` select from two or more options `check:` a boolean option `file:` file name `number:` real number `integer:` integer number	`type=text`
`label`	The label for the user interface widget.	`label="Account"`
`default`	Default value for this tag	`default="interactive"`
`text` type		
`mask`	Input mask `0:` ASCII digit permitted but not required. `9:` ASCII digit required. 0-9. `N:` ASCII alphanumeric character required. A-Z, a-z, 0-9. `n:` ASCII alphanumeric character permitted but not required.	`mask="09:09:09"`
`options` type		
`options`	Options to use, separated by the | character	`options="not_shared|shared"`
`check` type		
`checked`	Value of a `check` tag if checked.	`checked="enabled"`
`unchecked`	Value of a `check` tag if unchecked.	`unchecked="enabled"`
`integer` and `number` types		
`min`	Minimum value.	`min="0"`
`max`	Maximum value.	`max="100"`
`step`	Amount to step by when the up or down arrows are clicked.	`step="1"`
`decimals`	Number of decimal places.	`decimals="2"`
`suffix`	Display only suffix (will not be included in tag value).	`suffix="s"`
`prefix`	Display only prefix (will not be included in tag value).	`prefix="$"`

© 2015 Allinea Software Ltd.

file type		
mode	open-file: an existing file save-file: a new or existing file existing-directory: an existing directory open-files: one or more existing files, separated by spaces	mode="open-file"
caption	Window caption for file chooser.	caption="Select File"
dir	Initial directory for file chooser.	dir="/work/output"
filter	Restrict the files displayed in the file chooser to a certain file pattern.	filter="Text files (*.txt)"

Examples

```
# JOB_TYPE_TAG: {type=select,options=parallel| \
   serial,label="Job Type",default=parallel}

# WALL_CLOCK_LIMIT_TAG: {type=text,label="Wall Clock Limit", \
   default="00:30:00",mask="09:09:09"}

# NODE_USAGE_TAG: {type=select,options=not_shared| \
   shared,label="Node Usage",default=not_shared}

# ACCOUNT_TAG: {type=text,label="Account",global}
```

See the template files in {installation-directory} /templates for more examples.

To specify values for these tags click the *Edit Template Variables* button on the *Job Submission Options* page (See Figure 101 *Queuing Systems* above) or the *Run* window. You will see a window similar to the one below:

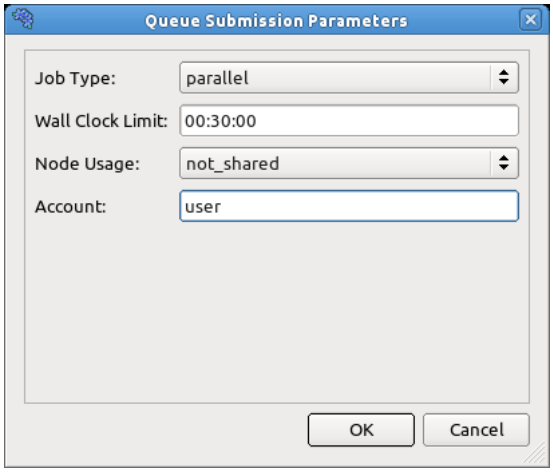


Figure 102: *Queue Parameters Window*

The values you specify are substituted for the corresponding tags in the template file when you run a job.

F.3 Specifying Default Options

A queue template file may specify defaults for the options on the Job Submission page so that when a user selects the template file these options are automatically filled in.

Name	Job Submission Setting	Example
`submit`	Submit command *Note: the command may include tags.*	qsub -n NUM_NODES_TAG -t WALL_CLOCK_LIMIT_TAG --mode script -A PROJECT_TAG
`display`	Display command *The output from this command is shown while waiting for a job to start.*	qstat
`job regexp`	Job regexp	(\d+)
`cancel`	Cancel command	qdel JOB_ID_TAG
`submit scalar`	Also submit scalar jobs through the queue	yes
`show num_procs`	Number of processes: Specify in Run window	yes
`show num_nodes`	Number of nodes: Specify in Run Window	yes
`show procs_per_node`	Processes per node: Specify in Run window	yes
`procs_per_node`	Processes per node: Fixed	16

Example

```
# submit: qsub -n NUM_NODES_TAG -t WALL_CLOCK_LIMIT_TAG \
   --mode script -A PROJECT_TAG
# display: qstat
# job regexp: (\d+)
# cancel: qdel JOB_ID_TAG
```

F.4 Launching

Ordinarily, your queue script will probably end in a line that starts `mpirun`with your target executable.

In a template file, this needs to be modified to run a command that will also launch the DDT/MAP backend agents.

Some methods to do this are mentioned in this section.

F.4.1 Using AUTO_LAUNCH_TAG

This is the easiest method, and caters for the majority of cases. Simply replace your `mpirun`command line with `AUTO_LAUNCH_TAG`. DDT/MAP will replace this with a command appropriate for your configuration (one command on a single line).

e.g. an `mpirun`line that looks like this:

```
mpirun -np 16 program_name myarg1 myarg2
```

simply becomes:

```
AUTO_LAUNCH_TAG
```

AUTO_LAUNCH_TAG is roughly equivalent to:

```
DDT_MPIRUN_TAG DDT_DEBUGGER_ARGUMENTS_TAG \
MPI_ARGUMENTS_TAG PROGRAM_TAG ARGS_TAG
```

A typical expansion is:

```
/opt/allinea/tools/bin/ddt-mpirun --ddthost login1,192.168.0.191 \
--ddtport 4242 --ddtsession 1 \
--ddtsessionfile /home/user/.allinea/session/login1-1 \
--ddtshareddirectory /home/user --np 64 \–
npernode 4 myprogram arg1 arg2 arg3
```

F.4.2 Using ddt-mpirun

If you need more control than is available using AUTO_LAUNCH_TAG, DDT/MAP also provides a drop-in mpirunreplacement that can be used to launch your job.

You should replace mpirunwith DDTPATH_TAG/bin/ddt-mpirun. For example, if your script currently has the line:

```
mpirun -np 16 program_name myarg1 myarg2
```

Then (for illustration only) the equivalent that DDT/MAP would need to use would be:

```
DDTPATH_TAG/bin/ddt-mpirun -np 16 program_name myarg1 myarg2
```

For a template script you use tags in place of the program name, arguments etc. so they can be specified in the GUI rather than editing the queue script each time:

```
DDTPATH_TAG/bin/ddt-mpirun -np NUM_PROCS_TAG \
EXTRA_MPI_ARGUMENTS_TAG DDTPATH_TAG/bin/ddt-debugger \
DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

See F.1 Queue Template Tags for more information on template tags.

F.4.3 MPICH 1 based MPI

If AUTO_LAUNCH_TAG or ddt-mpirun are not suitable, you can also use the following method for MPICH 1 based MPIs.

If your mpiruncommand line looks like:

```
mpirun -np 16 program_name myarg1 myarg2
```

You need to export the TOTALVIEW environment variable, and add the -tv parameter to mpirun. e.g.

```
export TOTALVIEW=DDTPATH_TAG/bin/ddt-debugger-mps
MPIRUN_TAG -np NUM_PROCS_TAG \
-tv PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

© 2015 Allinea Software Ltd.

F.4.4 Scalar Programs

If AUTO_LAUNCH_TAG isn't suitable, you can also use the following method to launch scalar jobs with your template script:

```
DDTPATH_TAG/bin/ddt-client DDT_DEBUGGER_ARGUMENTS_TAG \
PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

F.5 Using PROCS_PER_NODE_TAG

Some queue systems allow you to specify the number of processes, others require you to select the number of nodes and the number of processes per node. The software caters for both of these but it is important to know whether your template file and queue system expect to be told the number of processes (NUM_PROCS_TAG) or the number of nodes and processes per node (NUM_NODES_TAG and PROCS_PER_NODE_TAG). If these terms seem strange, see sample.qtf for an explanation of the queue template system.

F.6 Job ID Regular Expression

The *Regexp for job id* regular expression is matched on the output from your submit command. The first bracketed expression in the regular expression is used as the job ID. The elements listed in the table are in addition to the conventional quantifiers, range and exclusion operators.

Element	Matches
C	A character represents itself
\t	A tab
.	Any character
\d	Any digit
\D	Any non-digit
\s	White space
\S	Non-white space
\w	Letters or numbers (a word character)
\W	Non-word character

For example, your submit program might return the output job id j1128 has been submitted – one possible regular expression for retrieving the job id is id\s(.+)\shas.

If you would normally remove the job from the queue by typing job_remove j1128 then you should enter job_removeJOB_ID_TAG as the cancel command.

© 2015 Allinea Software Ltd.

Index

AIX, 159, 162, 172
Align Stacks, 59
Allinea DDT
    Getting Started, 20
    Installation, 12
    Introduction, 9
    Obtaining Help, 11
    Online Resources, 11
    Starting a program, 50
Allinea MAP
    Installation, 12
    Introduction, 10
    Obtaining Help, 11
    Online Resources, 11
    Starting, 123
Altix, 159
AMD
    OpenCL, 166
Apple, 18
ARM, 172
Attaching, 32, 109
    Choose Hosts, 32
    Command Line, 33
    Hosts File, 33

Backtrace, 59
Berkeley UPC, 165
Blue Gene/Q, 173
Bounds Checking, 90
Breakpoints, 51
    Conditional, 53
    Deleting, 53
    Saving, 54
Buffer Overflow, 42
Bull MPI, 161

CAPS HMPP, 112
Colour Scheme, 152
Complex Numbers, 69
Configuration, 26
    Site Wide, 146
Consistency Checking
    Heap, 92
Core FIles, 29
CPU branch, 143
CPU floating-point, 143
CPU floating-point vector, 143
CPU integer, 143
CPU integer vector, 143
CPU memory access, 143
Cray, 112, 171
Cray MPT, 164
Cray Native SLURM, 165, 171
Cray X, 164
Cray XK6, 171
Cross-Process Comparison, 79
Cross-Thread Comparison, 79
CUDA
    Breakpoints, 54
    CUDA Fortran, 113
    DDT: CUDA, 105
    GPU Debugging, 105
    Memory Debugging, 90
    NVIDIA, 105
    Running, 23

Data
    Changing, 72
Deadlock, 89
Disk read transfer, 143
Disk write transfer, 143

Editor, 152
End Session, 25
Environment Variables, 24

Fencepost Checking, 96
Font, 152
Fortran Modules, 68
Function Listing, 42

GPU, 105
    Attaching, 109
    GPU Language Support, 112

Heap Overflow, 96
HMPP, 112
Hotkeys, 50
HP MPI, 161

Inf, 63
Input, 83, 131
Installation, 12
    Linux, 12
    Mac OS X, 14
    Text-mode Install, 13
    Windows, 14
Intel Compiler, 24, 163, 167
Intel Message Checker, 161
Intel MPI, 161
    MPMD, 28

remote-exec, 25
Irix, 164

Job Submission, 33, 132
Cancelling, 33, 132
Custom, 34
Regular Expression, 33, 132, 193
Jump To Line, 41
Double Clicking, 45

Licensing
Floating Licences, 15
Licence Files, 15
Licence Server, 154
Purchasing, 10
Single Process Licence, 26, 131
Loadleveler, 159
Log file, 187

Mac OS X, 20
Macros, 68
Main Window
Overview, 38
Manual Launch
ddt-client, 27
Debugging Multi-Process Non-MPI programs, 27
Memory Debugging, 90
Configuration, 90
Enabling, 24
Memory Statistics, 98
mprotect fails, 184
Memory Leak, 42
Memory Usage, 97, 142
Message Queues, 87
Moab, 159
MOM nodes, 164
MPI
Function Counters, 102
History/Logging, 101
MPI Rank, 45
MPI Ranks, 81
mpirun, 23
Running, 23
Troubleshooting, 180
MPI bytes sent / received, 142
MPI call duration, 142
MPI point-to-point / collective operations, 143
MPI_Init
remote-exec, 25
MPICH, 130
MPICH 1
remote-exec, 25
MPICH 2
MPMD, 28
remote-exec, 25
MPICH 3, 162
MPMD, 28
remote-exec, 25
mpirun
remote-exec, 25
MPMD
Compatibility Mode, 29
Intel MPI, 28
MPICH 2, 28
MPICH 3, 28
remote-exec, 130
Running, 28
MVAPICH, 162
MVAPICH 2, 163

nvcc, 105
NVIDIA Tegra 2, 172

Obtaining Help, 11
Online Resources, 11
Open MPI, 163
MPMD, 28
Compatibility Mode, 29
OpenACC, 112
OpenCL, 105
OpenGL, 78, 79
OpenMP
OMP_NUM_THREADS, 26
Running, 23, 26
Oracle Grid Engine, 159

Parallel Stack View, 60
PBS, 159
PGI Accelerators, 113
Plugins
Enabling, 24
Pointers, 72
Portland Group, 169
Pretty Printers, 70
Process Groups, 45
Deleting, 45
Programming errors, 152

Queue Submission, 33
Cancelling, 33

Raw Command, 82
Receive queue, 89
Registers

Viewing, 81
Remote Client, 16
Configuration, 16
Installation
Mac OS X, 14
Windows, 14
Multiple Hops, 17
Remote Script, 17
remote-exec
Required, 25
Restarting, 50
Running
MPMD, 28
Scalar, 26

Scalar
Running, 26
Search, 40, 41
Send queue, 89
Session
Saving, 39
Session Menu, 50
SGI, 164
SGI MPT
remote-exec, 25
Shared Arrays, 72
Signal Handling, 63
Divisions by zero, 63
Floating Point Exception, 63
Segmentation fault, 63
SIGFPE, 63
SIGILL, 63
SIGPIPE, 63
SIGSEGV, 63
SIGUSR1, 64
SIGUSR2, 64
Single Stepping, 50
SLURM, 165
SMP
Performance, 179
Source Code, 39, 62
Editing, 43
Missing Files, 40
Searching, 40, 41
Sparkline, 79
Stack Frame, 59
Standard Error, 83, 137
Standard Output, 83, 137
Starting MAP, 123
Static Analysis, 42
Static checking, 152
Step Threads Together, 48
Stopping, 50
Synchronizing Processes, 54

Tab size, 152
TORQUE, 159
Tracepoints, 56

Unexpected queue, 89
UPC, 71

Variables, 43, 66
Searching, 40, 41
Unused Variables, 42
Visualize Whitespace, 152
VNC, 18

Warning Symbols, 42
Watchpoints, 55
Welcome Page, 20
Welcome Screen, 123

X forwarding, 18
X11, 179
XK6, 171

© 2015 Allinea Software Ltd.